

# Micro Services Architecture

Part 1 : Infrastructure Comparison &  
Design Styles (DDD, Event Sourcing / CQRS, Functional Reactive Programming)  
Araf Karsh Hamid – Co Founder / CTO, MetaMagic Global Inc., NJ, USA

A **Micro Service** will have its own Code Pipeline for build and deployment functionalities and it's scope will be defined by the Bounded Context focusing on the Business Capabilities and the interoperability between Micro Services will be achieved using message based communication.

# Pioneers in Microservices Implementation

---



---

## New Entrants

---



# Agenda

---

1

## Architecture Styles

- Pros and Cons
- Micro Services Characteristics
- Monolithic Vs. Micro Services Architecture
- SOA Vs. Micro Services Architecture
- App Scalability Based on Micro Services
- Hexagonal Architecture

3

## Scalability

- CAP Theorem
- Distributed Transactions : 2 Phase Commit
- SAGA Design Pattern
- Scalability Lessons from EBay
- Design Patterns
- References

2

## Design Styles

- Design Patterns
- Domain Driven Design
- Event Sourcing & CQRS
- Functional Reactive Programming

# Pros and Cons

## Pros

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

## Cons

1. Adds Complexity
2. Skillset shortage
3. Confusion on getting the right size
4. Team need to manage end-to-end of the Service (From UI to Backend to Running in Production).

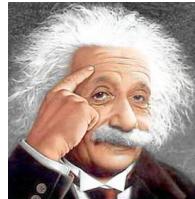
# Micro Services Characteristics

By James Lewis and Martin Fowler

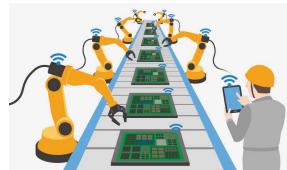
**Components**  
via  
Services



**Smart  
Endpoints**  
& Dumb Pipes



**Infrastructure  
Automation**



The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.  
**Alan Kay, 1998 email to the Squeak-dev list**

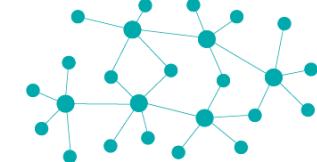
Organized around  
**Business  
Capabilities**



**Products**  
NOT  
Projects



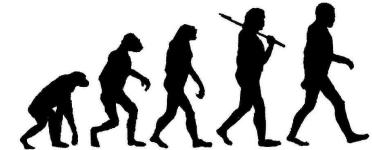
**Decentralized  
Governance &  
Data Management**



Design for  
**Failure**



**Evolutionary  
Design**



Modularity ... is to a technological economy what the division of labor is to a manufacturing one.  
**W. Brian Arthur,**  
author of *e Nature of Technology*

We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration. —  
**Werner Vogels, CTO, Amazon Web Services**

# Micro Services System Design Model

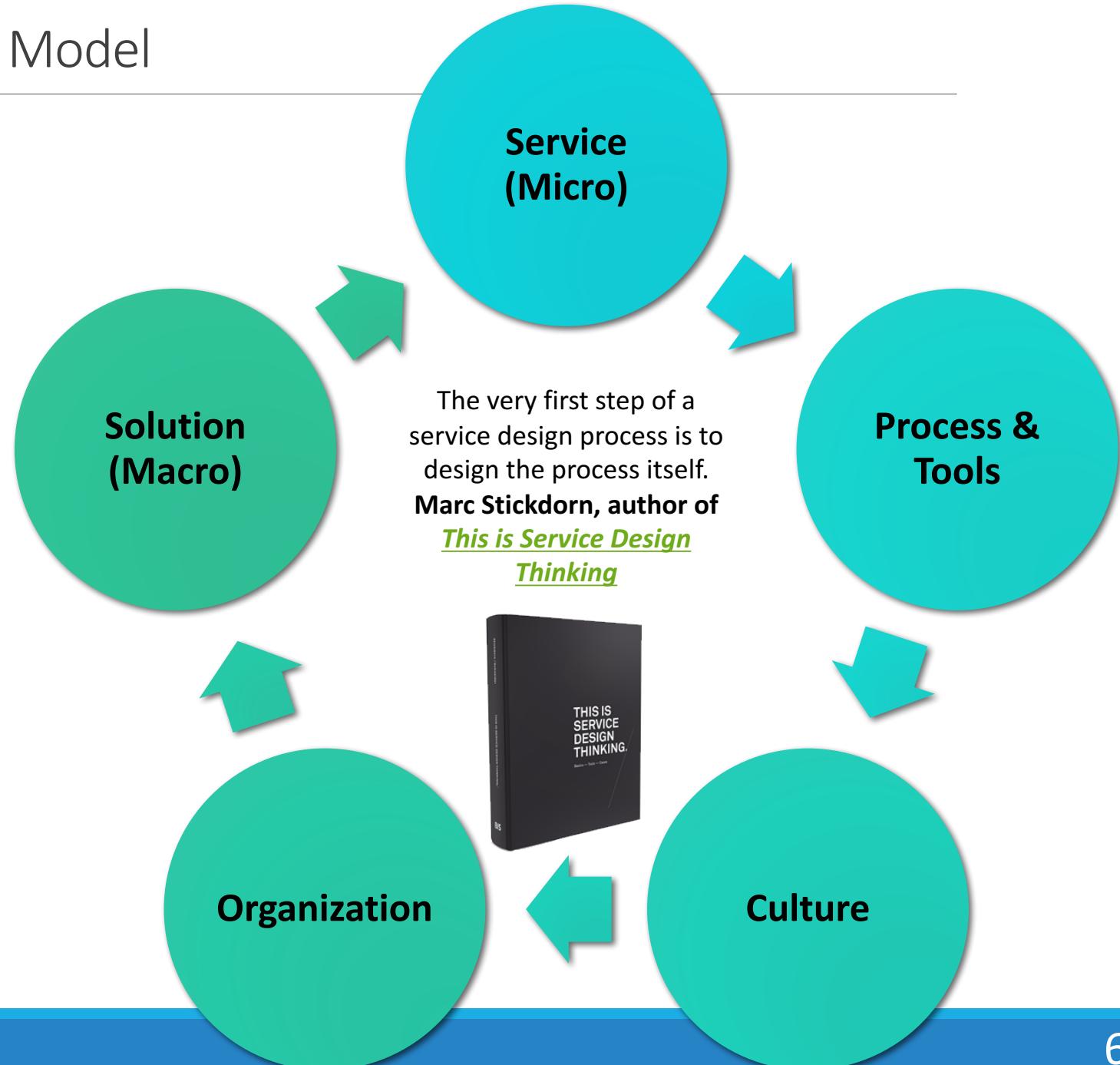
**Service:** Focuses on a specific Business Capability

**Process & Tools:** Development, Code Deployment, Maintenance and Product Management

**Culture:** A Shared set of values, beliefs by everyone. **Ubiquitous Language in DDD is an important aspect of Culture.**

**Organization:** Structure, Direction of Authority, Granularity, Composition of Teams.

**Solution:** Coordinate all inputs and outputs of multiple services. Macro level view of the system allows the designer to focus more on desirable system behavior.



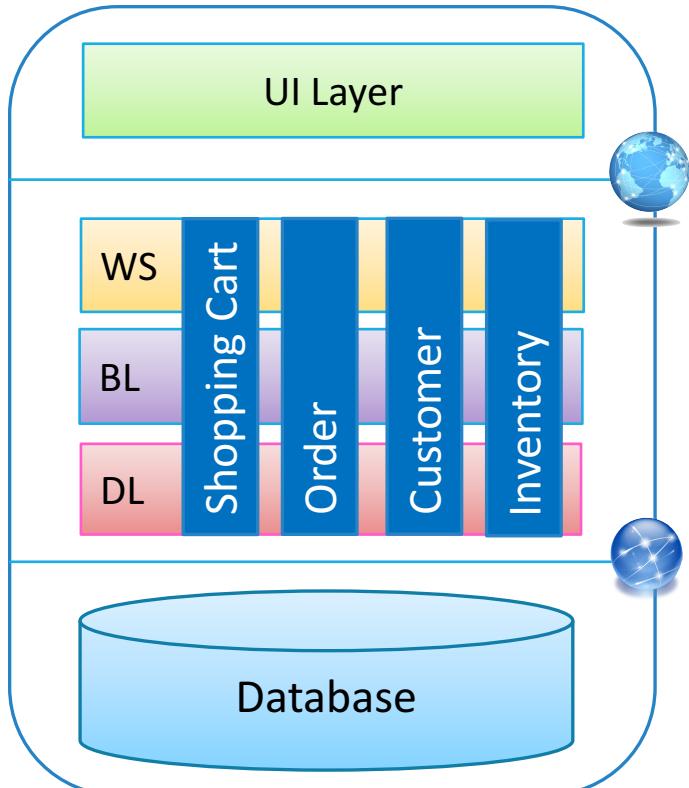
# Monolithic vs. Micro Services Example



Existing aPaaS vendors creates Monolithic Apps.

This 3 tier model is obsolete now.

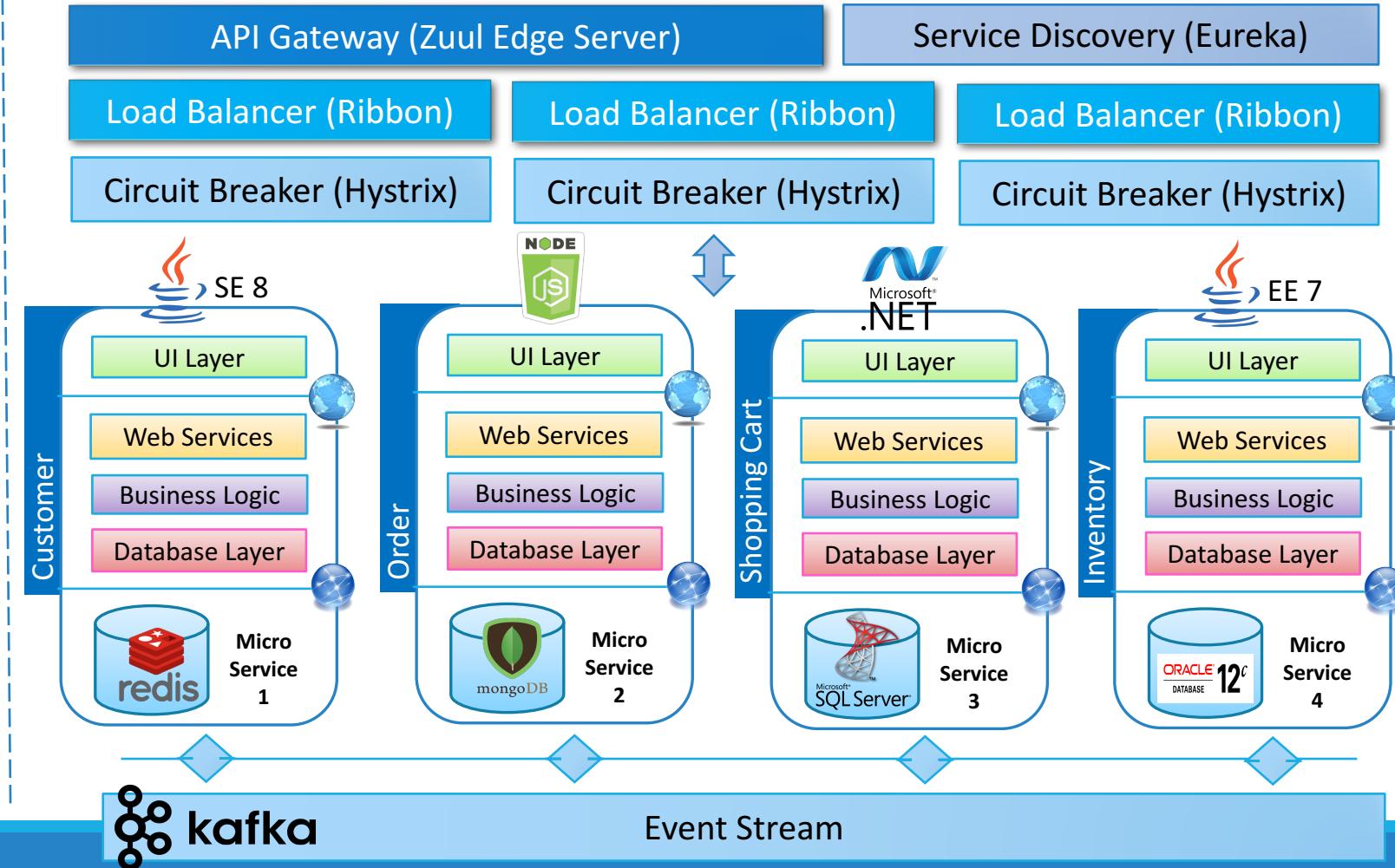
Source: Gartner Market Guide for Application Platforms Nov 23, 2016



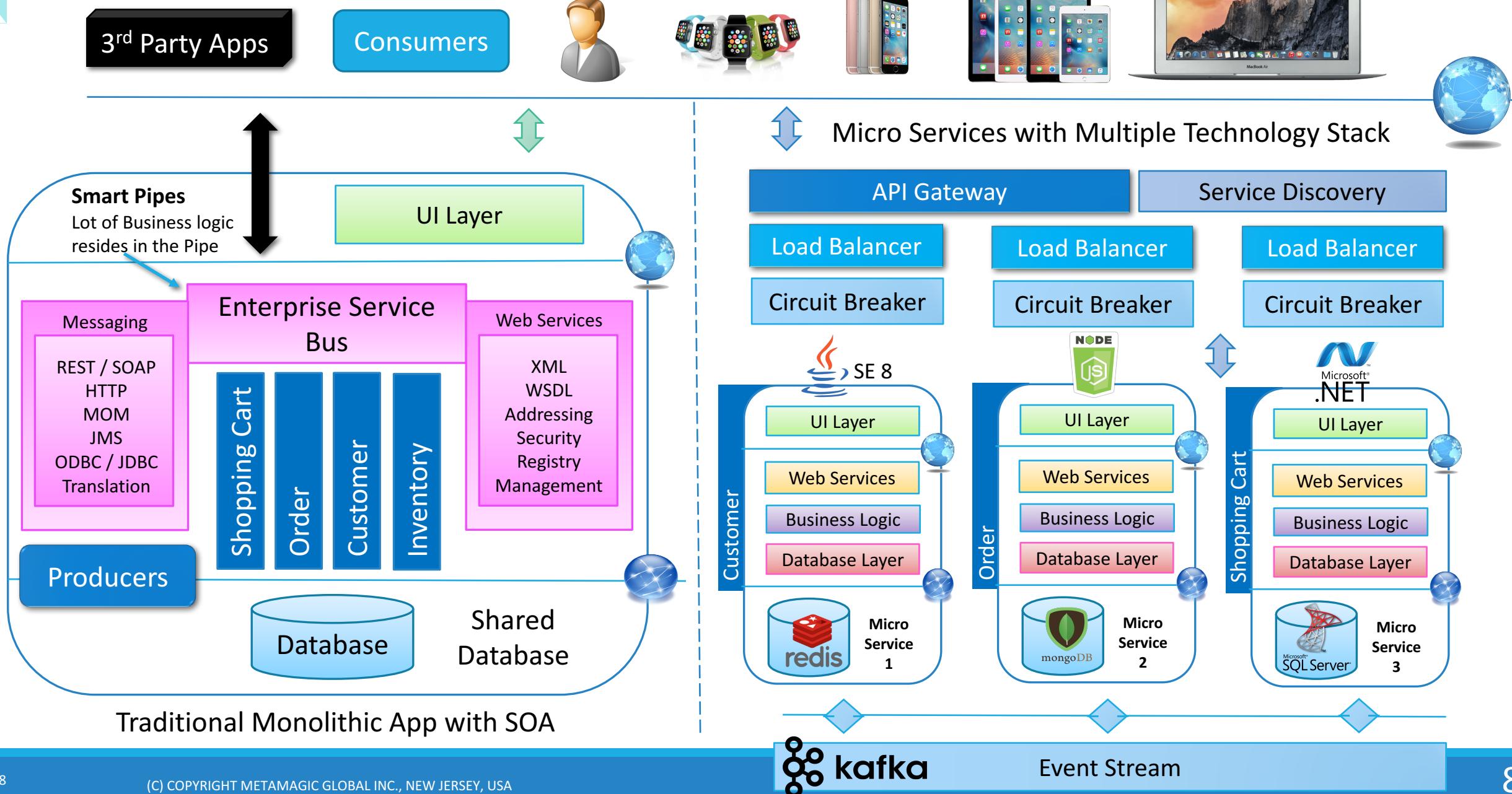
Traditional Monolithic App using Single Technology Stack



## Micro Services with Multiple Technology Stack



# SOA vs. Micro Services Example

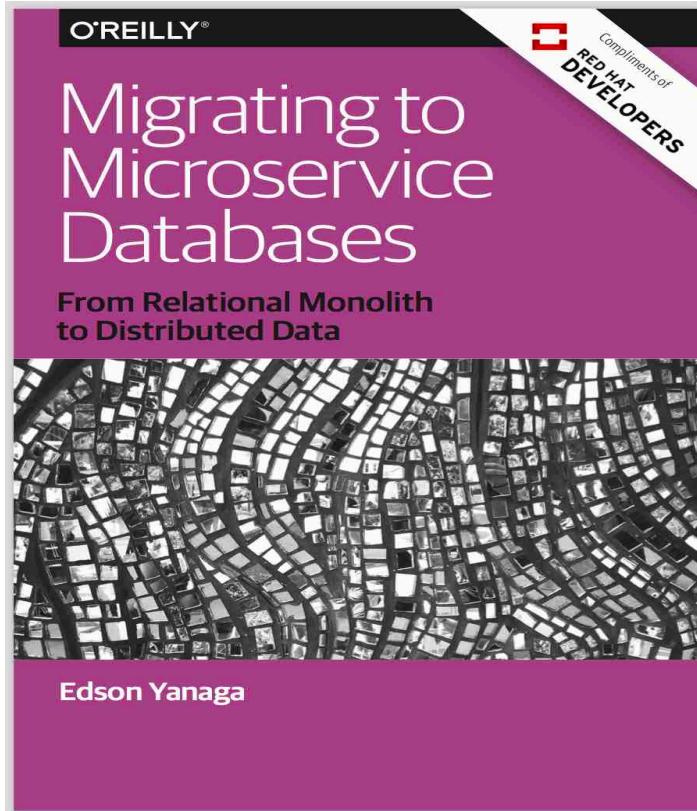


# API Gateway Features (Zuul)

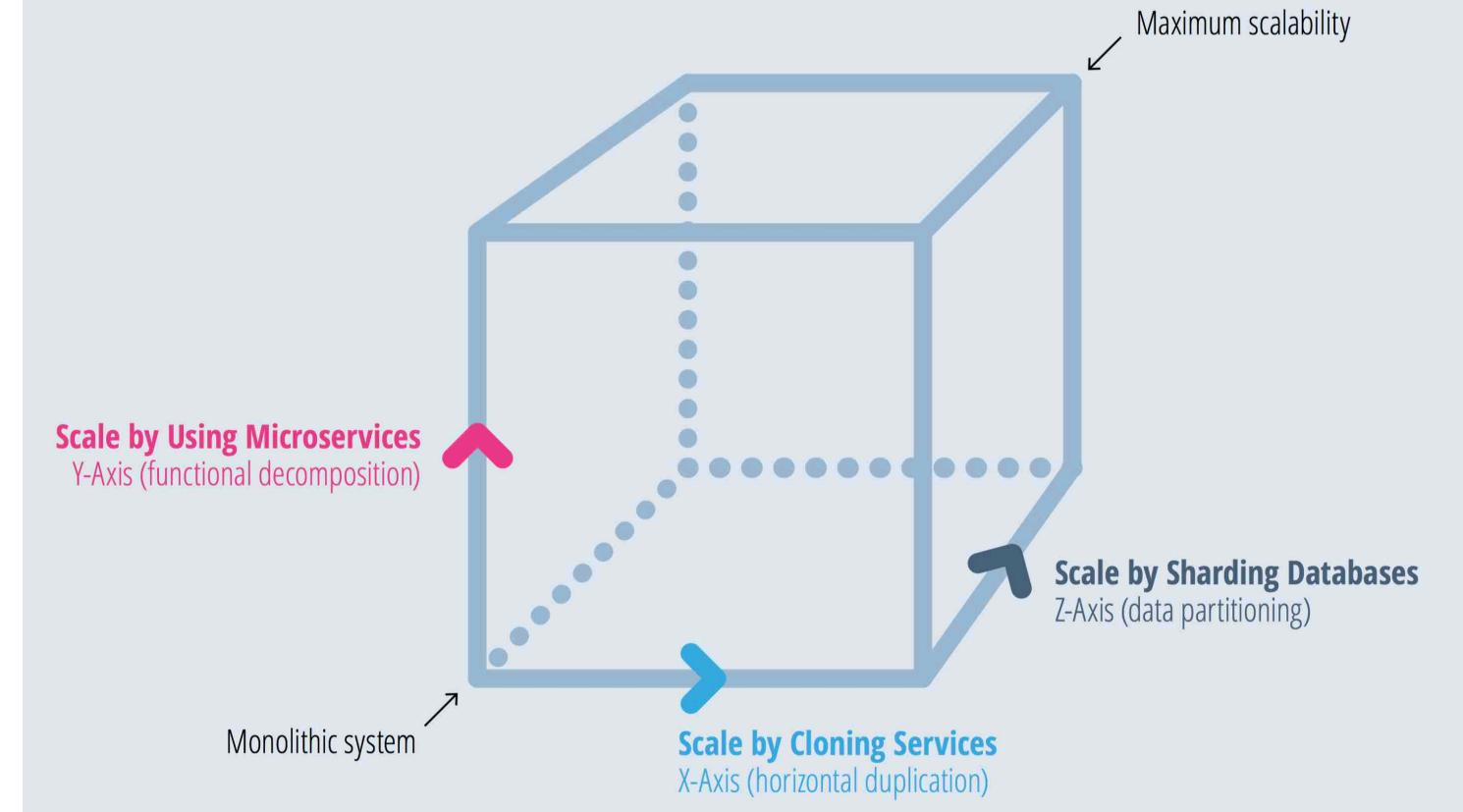
- **Authentication and Security:** identifying authentication requirements for each resource.
- **Insights and Monitoring:** tracking meaningful data and statistics.
- **Dynamic Routing:** dynamically routing requests to different backend..
- **Stress Testing:** gradually increasing the traffic.
- **Load Shedding:** allocating capacity for each type of request and dropping requests.
- **Static Response handling:** building some responses directly.
- **Multi region Resiliency:** routing requests across AWS regions.
- **Hystrix** is used to wrap calls to our origins, which allows us to shed and prioritize traffic when issues occur.
- **Ribbon** is our client for all outbound requests from Zuul, which provides detailed information into network performance and errors, as well as handles software load balancing for even load distribution.
- **Turbine** aggregates fine-grained metrics in real-time so that we can quickly observe and react to problems.
- **Archaius** handles configuration and gives the ability to dynamically change properties.

Source: [https://dzone.com/articles/spring-cloud-netflix-zuul-edge-serverapi-gatewayga](https://dzone.com/articles/spring-cloud-netflix-zuul-edge-server-api-gatewayga)

# App Scalability based on micro services architecture



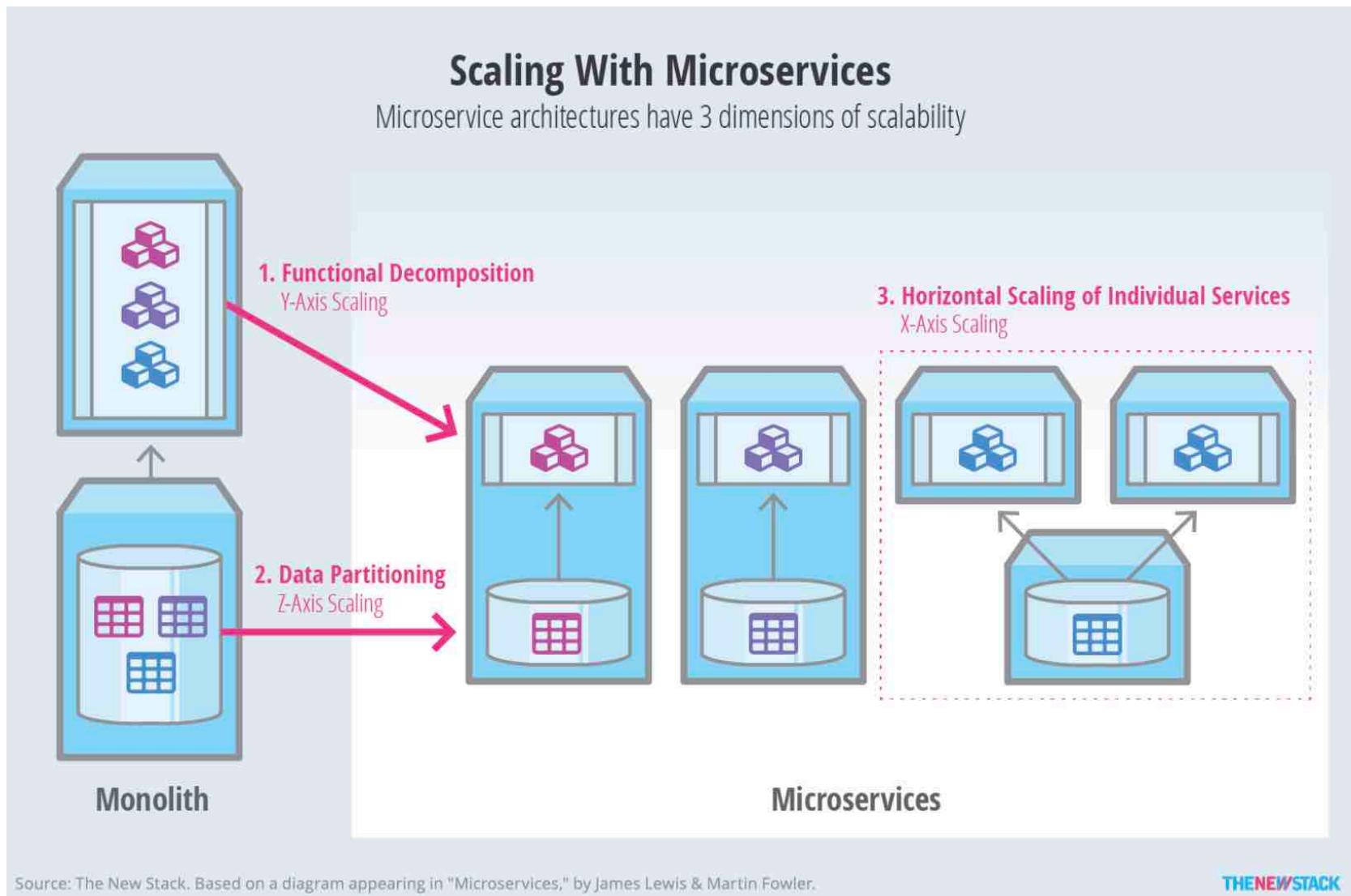
## The Scale Cube and Microservices: 3 Dimensions to Scaling



Source: The NewStack. Based on the Art of Scalability by By Martin Abbot & Michael Fisher

# Scale Cube and Micro Services

1. Functional Decomposition
2. Avoid locks by Database Sharding
3. Cloning Services



# Hexagonal Architecture

## Ports & Adapters

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

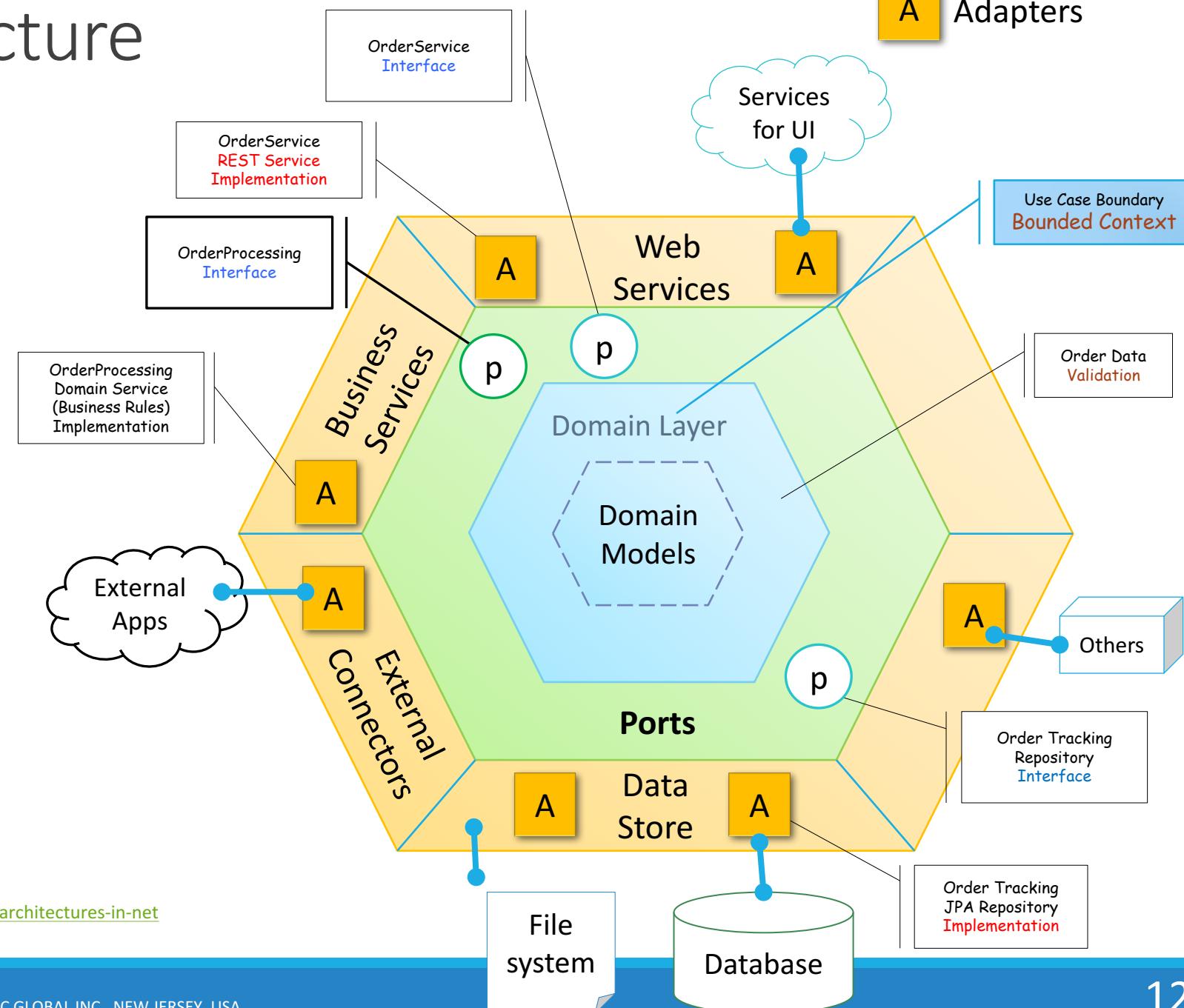
The notion of a “port” invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.

- Reduces Technical Debt
- Dependency Injection
- Auto Wiring

Source : <http://alistair.cockburn.us/Hexagonal+architecture>

<https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net>

A Adapters



# Micro Services Workshop Setup

Users



HTTP Server  
All UI Code is bundled

Micro Services with Multiple Technology Stack

API (Zuul) Gateway

LB = Ribbon

CB = Hystrix



UI Layer

Web Services

Business Logic

Database Layer

LB = Ribbon

CB = Hystrix



UI Layer

Web Services

Business Logic

Database Layer

LB = Ribbon

CB = Hystrix



UI Layer

Web Services

Business Logic

Database Layer

LB = Ribbon

CB = Hystrix



UI Layer

Web Services

Business Logic

Database Layer

Config Server (Spring)

Service Discovery (Eureka)

Product

Micro Service 1  
With 4 node cluster



Shopping Cart



Micro Service 2

Order

Micro Service 3  
With 2 node  
Cluster



Customer

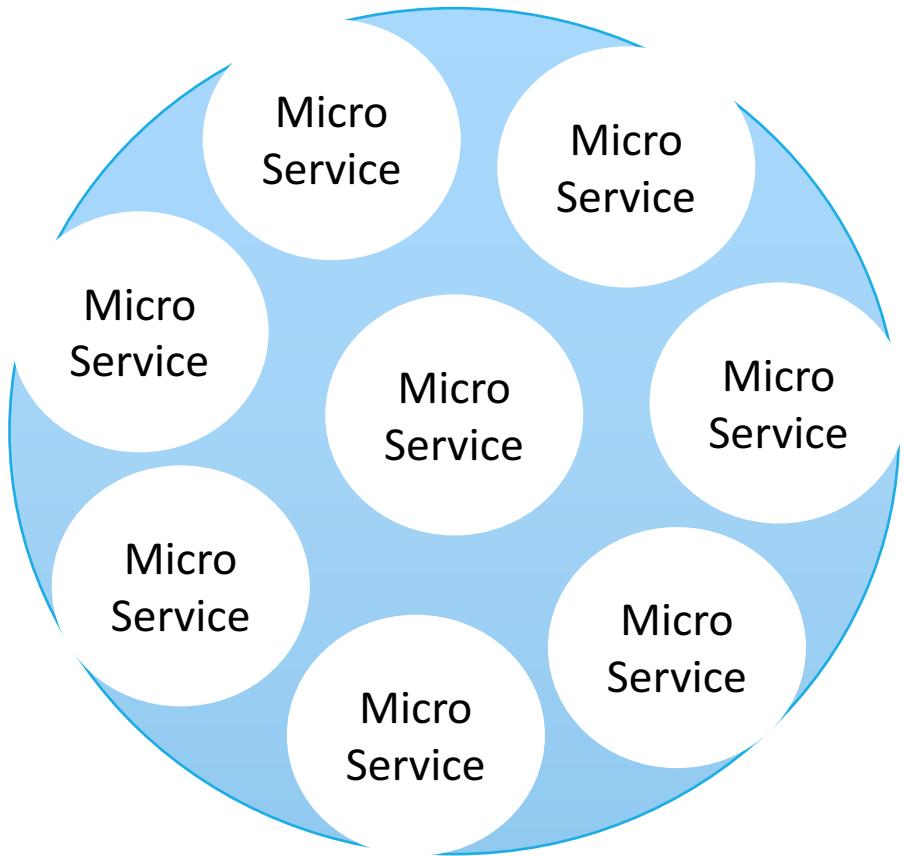


Virtual Private Network



Event Stream

# Summary – Micro Services Intro



Martin Fowler – Micro Services Architecture  
<https://martinfowler.com/articles/microservices.html>

Dzone – SOA vs Micro Services : <https://dzone.com/articles/microservices-vs-soa-2>

## Key Features

1. Small in size
2. Messaging-enabled
3. Bounded by contexts
4. Autonomously developed
5. Independently deployable
6. Decentralized
7. Language-agnostic
8. Built and released with automated processes

## Benefits

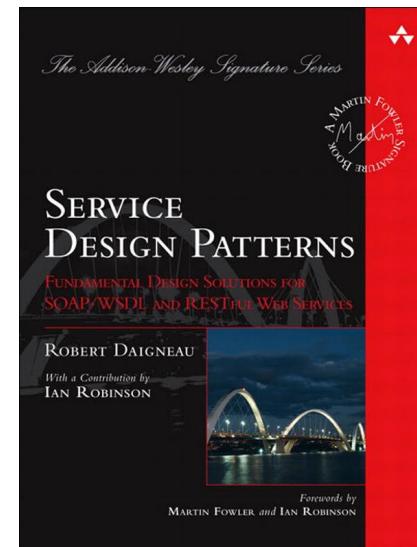
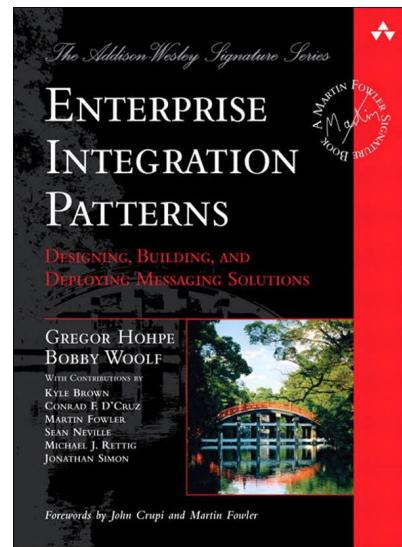
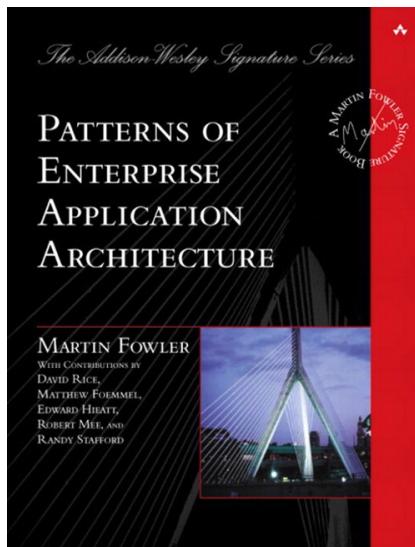
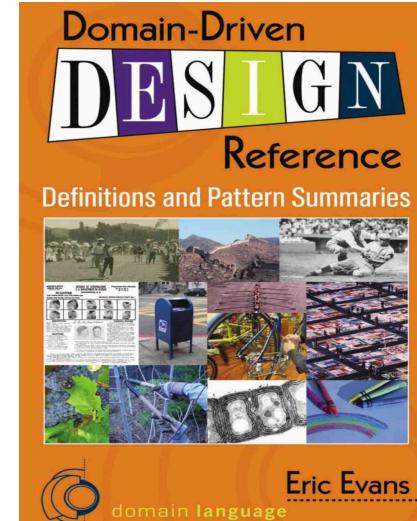
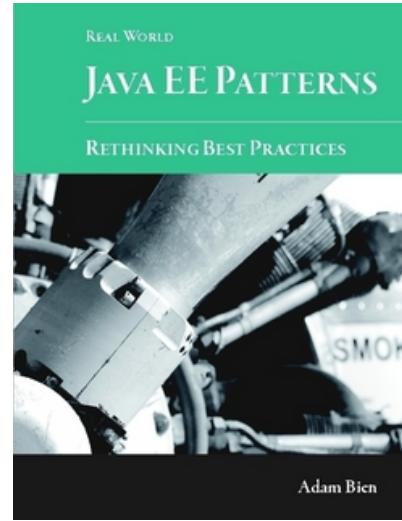
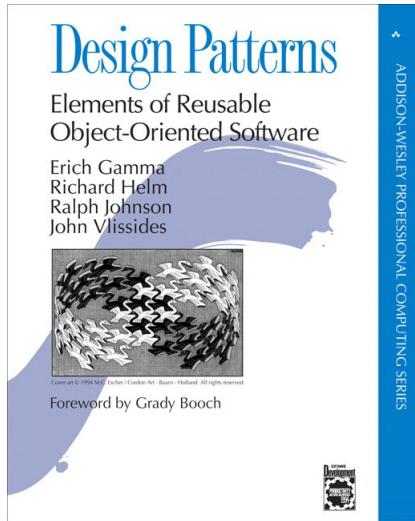
1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

# Design Styles

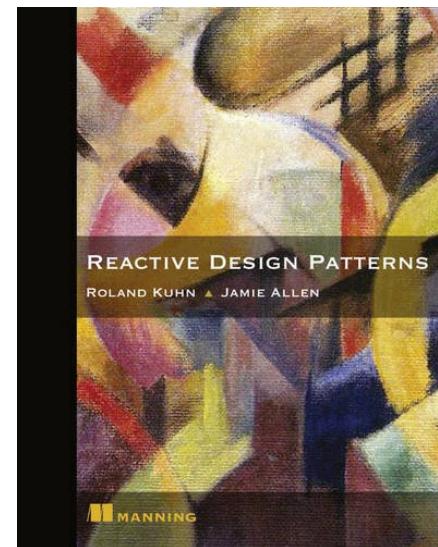
- Domain Driven Design
  - Understanding Requirement Analysis
  - Bounded Context
  - Context Map
  - Aggregate Root
- Event Sourcing & CQRS
  - CRUD
  - ES and CQRS
  - Event Sourcing Example
- Functional Reactive Programming
  - 4 Building Blocks of RxJava
  - Observable and Observer Design Pattern
  - Comparison : Iterable / Streams / Observable
  - Design Patterns : Let it Crash / SAGA

It's not necessary that you need to use all these patterns. You will be using these based on your technical requirement

# Design Patterns – Holy Grail of Developers

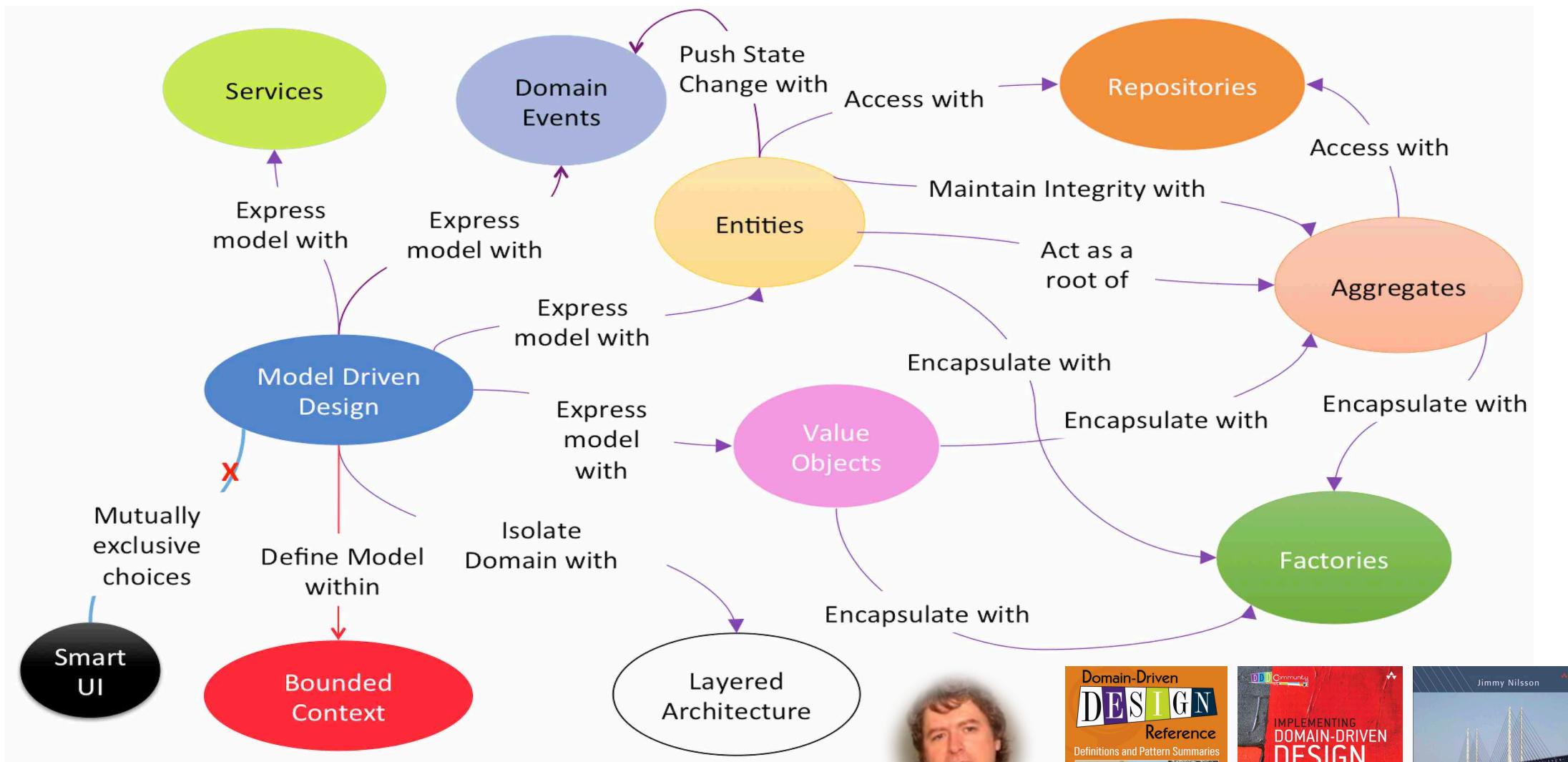


Design Patterns are solutions to general problems that software developers faced during software development.

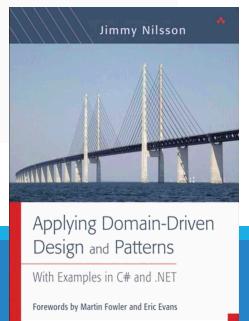
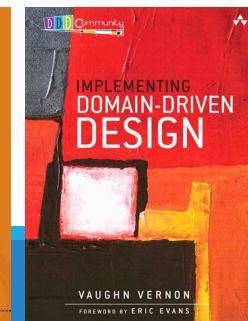
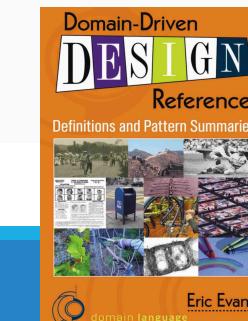


# Domain Driven Design

4/16/18



Source: Domain-Driven Design Reference by Eric Evans



# Ubiquitous Language : Understanding Requirement Analysis using DDD

**Ubiquitous  
Language**

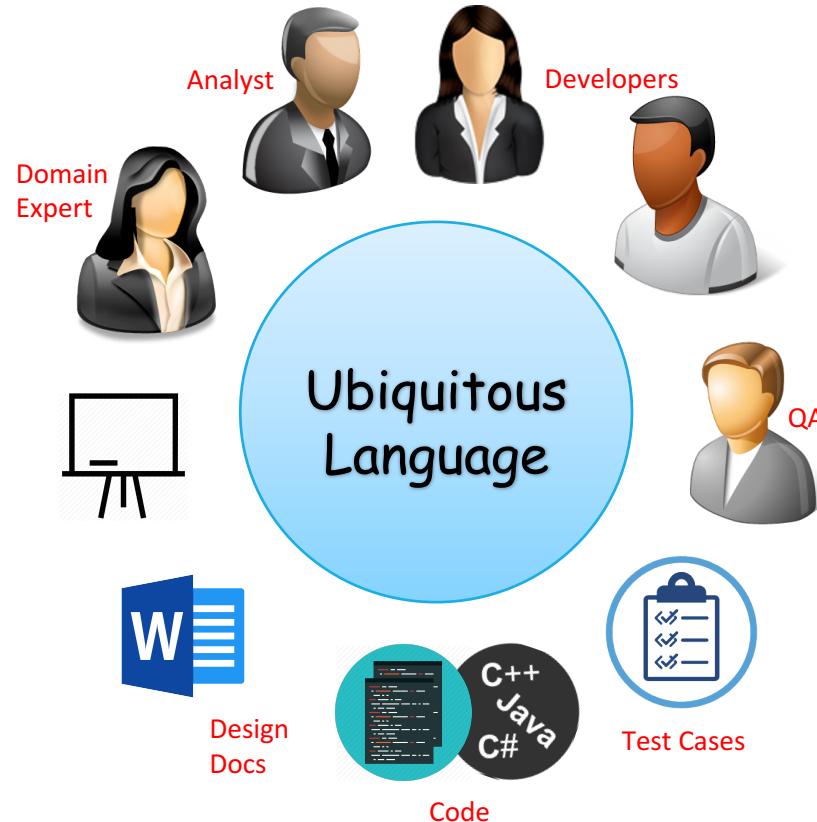
Vocabulary shared by  
all involved parties

Used in all forms of spoken /  
written communication

## Restaurant Context – Food Item :

Eg. Food Item (Navrathnakurma) can have different meaning or properties depends on the context.

- In the Menu Context it's a Veg Dish.
- In the Kitchen Context it's a recipe.
- And in the Dining Context it will have more info related to user feed back etc.



## Ubiquitous Language using BDD

As an insurance Broker  
I want to know who my Gold Customers are  
So that I sell more

Given Customer John Doe exists

When he buys insurance ABC for \$1000 USD

Then He becomes a Gold Customer

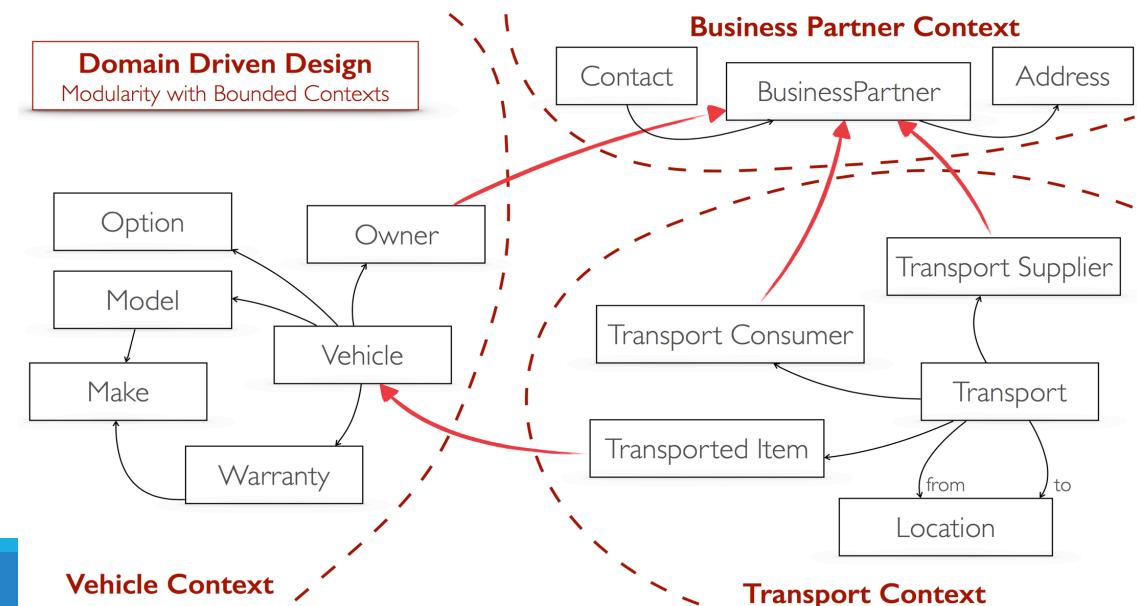
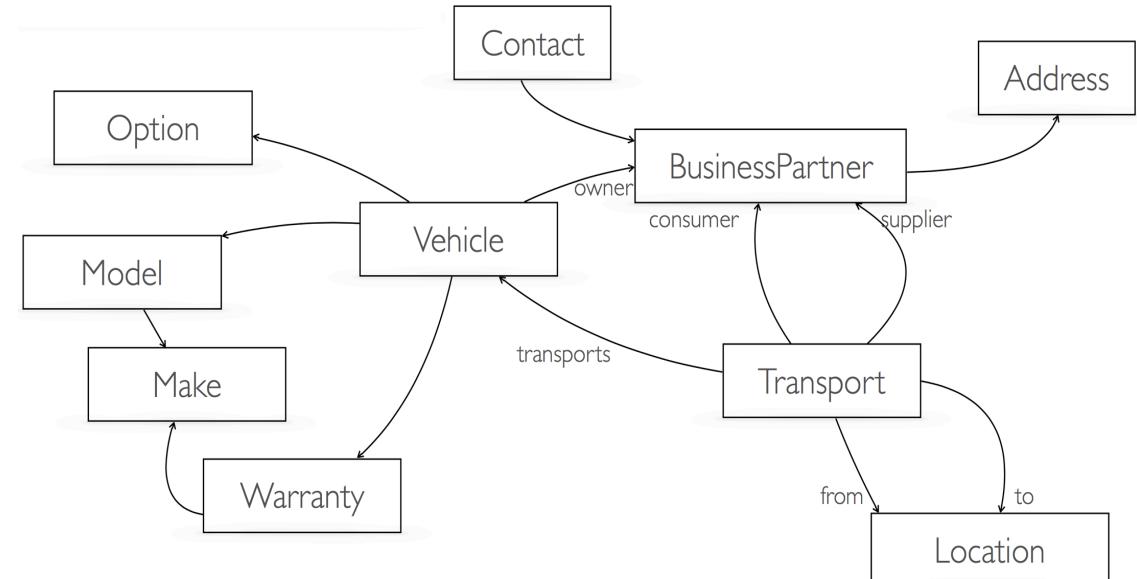
BDD – Behavior Driven Development

# Understanding Requirement Analysis using DDD

**Bounded Context**

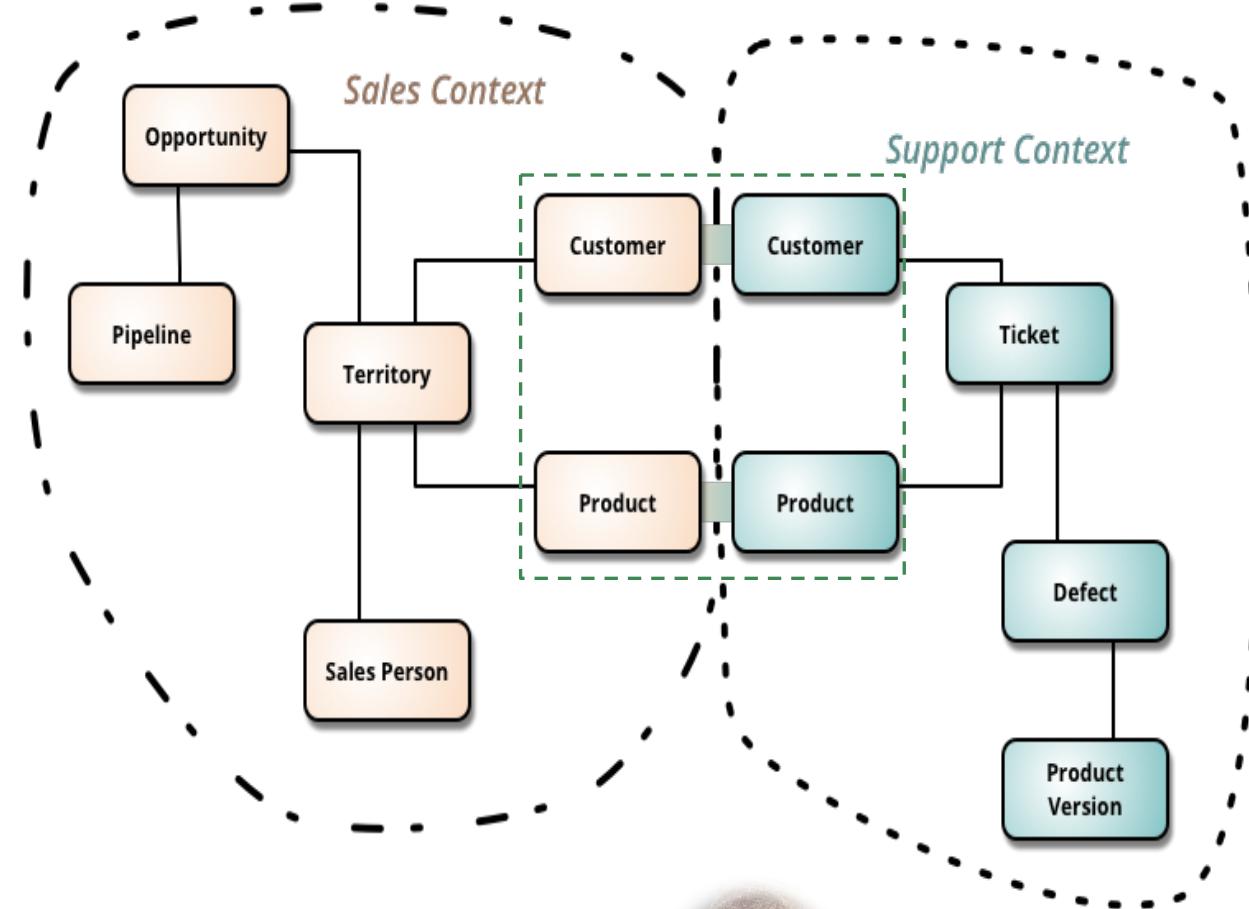
Areas of the domain treated independently

Discovered as you assess requirements and build language



# DDD : Understanding Bounded Context

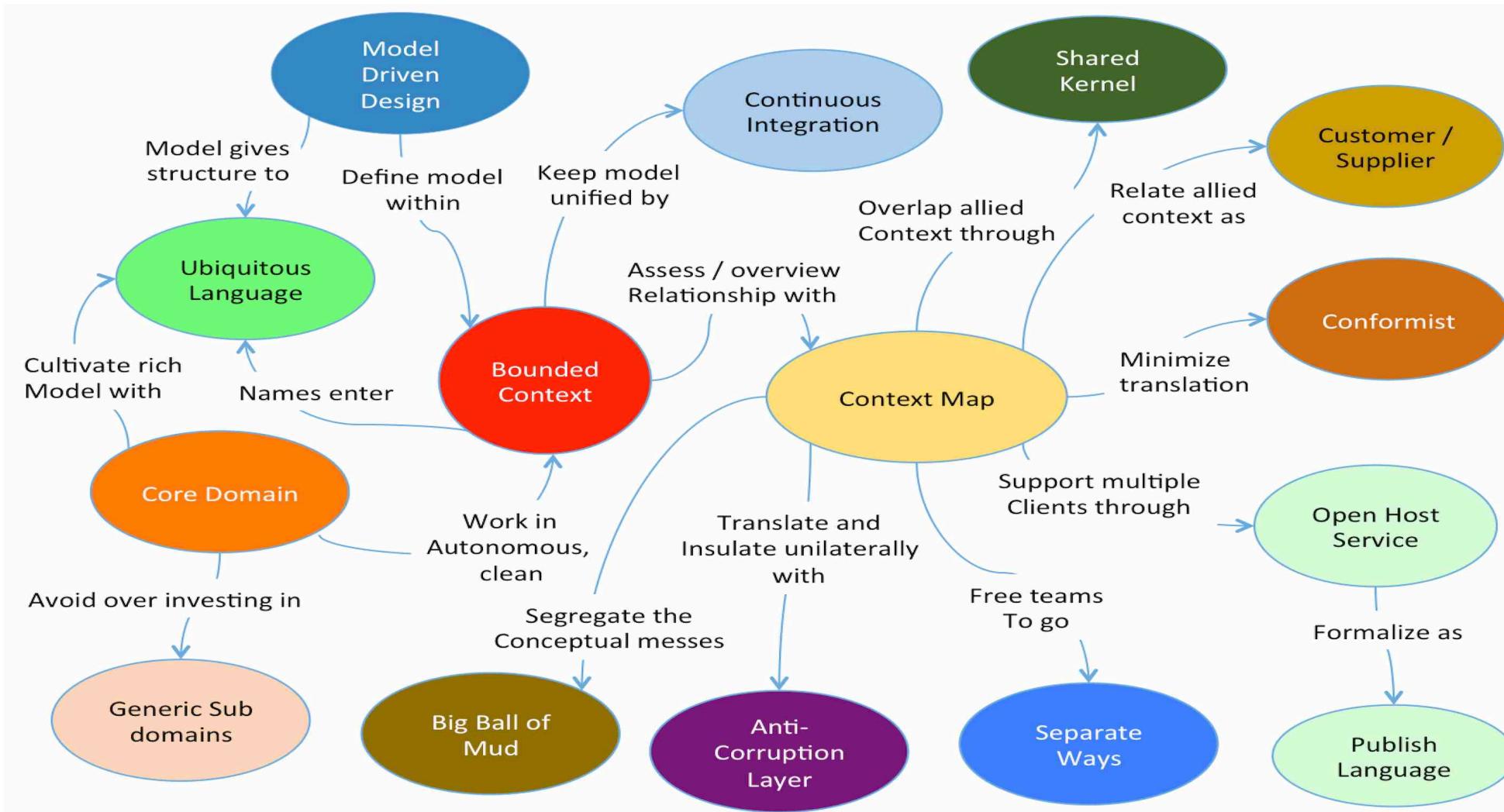
- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.
- Bounded Contexts have both unrelated concepts
  - Such as a support ticket only existing in a customer support context
  - But also **share** concepts such as **products** and **customers**.
- Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration.



Source: BoundedContext By Martin Fowler :  
<http://martinfowler.com/bliki/BoundedContext.html>



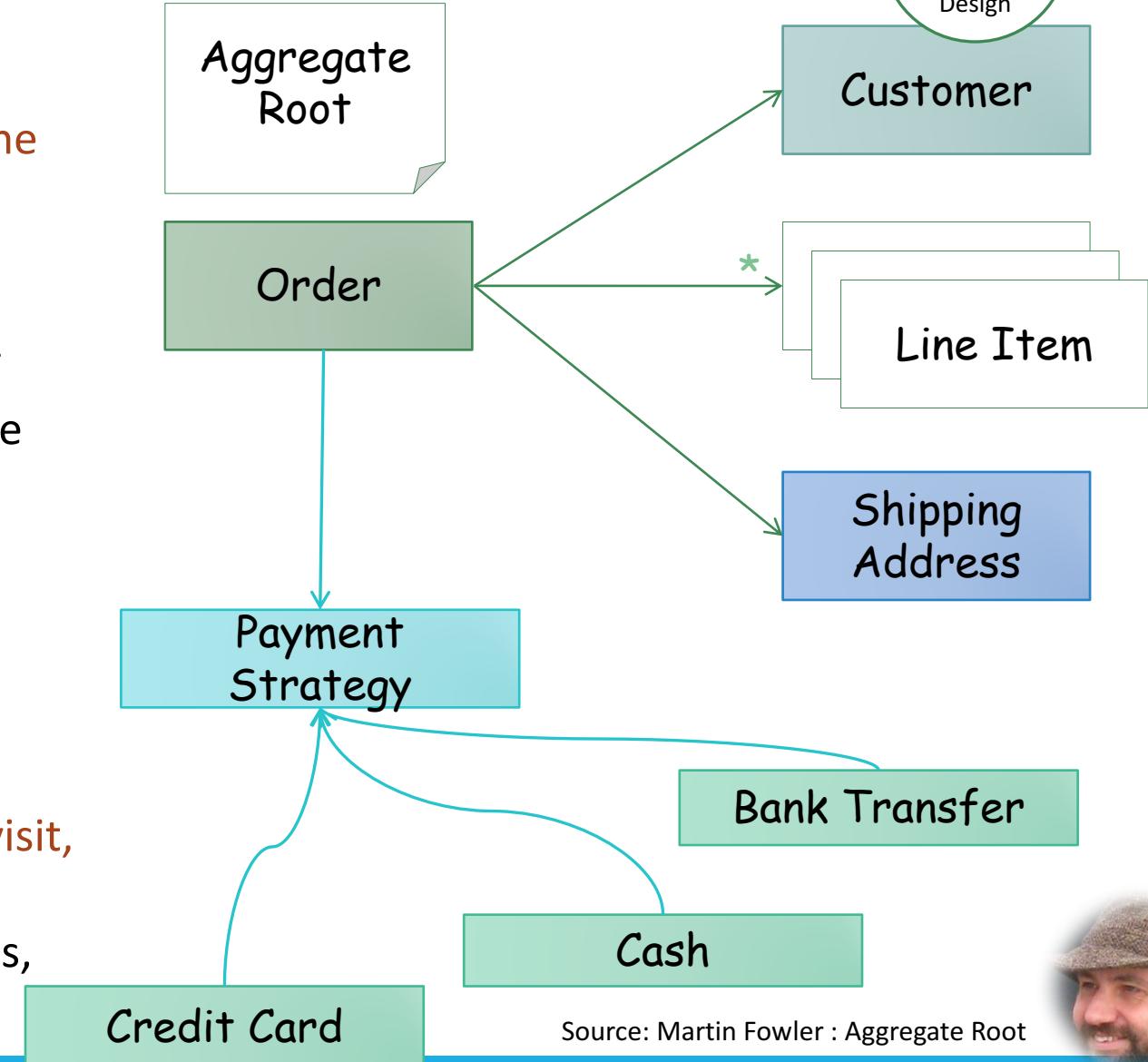
# DDD : Context Map



Source: Domain-Driven Design Reference by Eric Evans

# Understanding Aggregate Root

- An aggregate will have one of its component objects be the aggregate root. **Any references from outside the aggregate should only go to the aggregate root.** The root can thus ensure the integrity of the aggregate as a whole.
- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.
- Aggregates are sometimes confused with collection classes (lists, maps, etc.).
- Aggregates are **domain concepts** (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields.

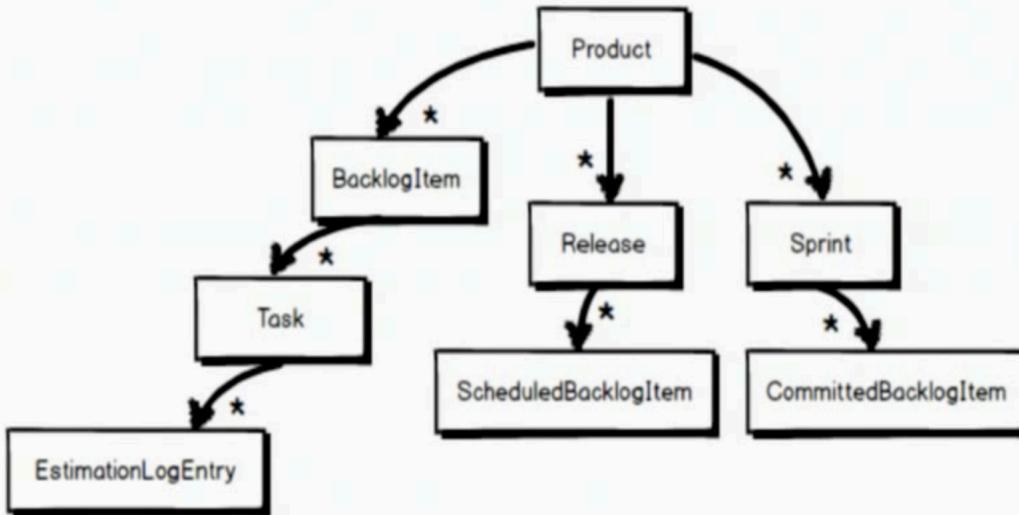


Source: Martin Fowler : Aggregate Root



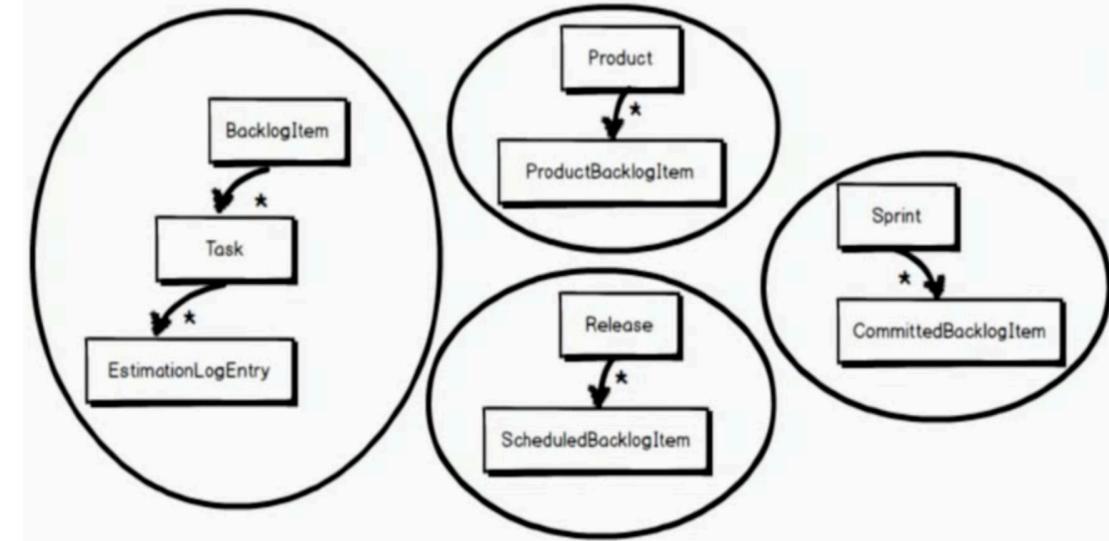
# Designing and Fine Tuning Aggregate Root

## Aggregate Root - #1



Super Dense Single Aggregate Root  
Results in Transaction concurrency issues.

## Aggregate Root - #2



Super Dense Aggregate Root is split into 4  
different smaller Aggregate Root in the 2<sup>nd</sup>  
Iteration.

Working on different design models helps the developers to come up with best possible design.

Source : Effective Aggregate Design Part 1/2/3 : Vaughn Vernon  
[http://dddcommunity.org/wp-content/uploads/files/pdf\\_articles/Vernon\\_2011\\_1.pdf](http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf)

# Rules for Building Aggregate Roots

1. Protect True Invariants in Consistency Boundaries. This rule has the **added implication that you should modify just one Aggregate instance in a single transaction**. In other words, when you are designing an Aggregate composition, plan on that representing a transaction boundary.
2. **Design Small Aggregates.** The smallest Aggregate you can design is one with a single Entity, which will serve as the Aggregate Root.
3. Reference Other Aggregates **Only By Identity**.
4. Use **Eventual Consistency** Outside the Consistency Boundary. This means that **ONLY ONE Aggregate instance will be required to be updated in a single transaction**. All other Aggregate instances that must be updated as a result of any one Aggregate instance update can be updated within some time frame (**using a Domain Event**). The business should determine the allowable time delay.
5. Build **Unidirectional Relationship** from the Aggregate Root.

# Data Transfer Object vs. Value Object

A small simple object, like money or a date range, whose equality isn't based on identity.

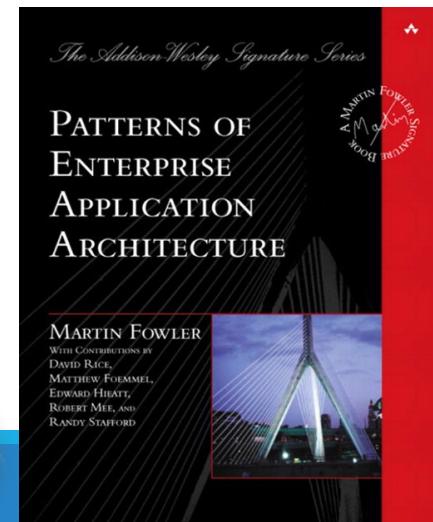
Data Transfer Object	Value Object
A DTO is just a data container which is used to transport data between layers and tiers.	A Value Object represents itself a fix set of data and is similar to a Java enum.
It mainly contains of attributes and it's a serializable object.	A Value Object doesn't have any identity, it is entirely identified by its value and is immutable.
DTOs are anemic in general and do not contain any business logic.	A real world example would be Color.RED, Color.BLUE, Currency.USD

486  
P of EAA

## Java EE 7 Retired the DTO

In Java EE the RS spec became the de-facto standard for remoting, so the implementation of serializable interface is no more required. To transfer data between tiers in Java EE 7 you get the following for FREE!

1. JAXB : Offer JSON / XML serialization for Free.
2. Java API for JSON Processing – Directly serialize part of the Objects into JSON



# DTO – Data Transfer Object

An object that carries data between processes in order to reduce the number of method calls.

**Problem:** How do you preserve the simple semantics of a procedure call interface without being subject to the latency issues inherent in remote communication?

## Benefits

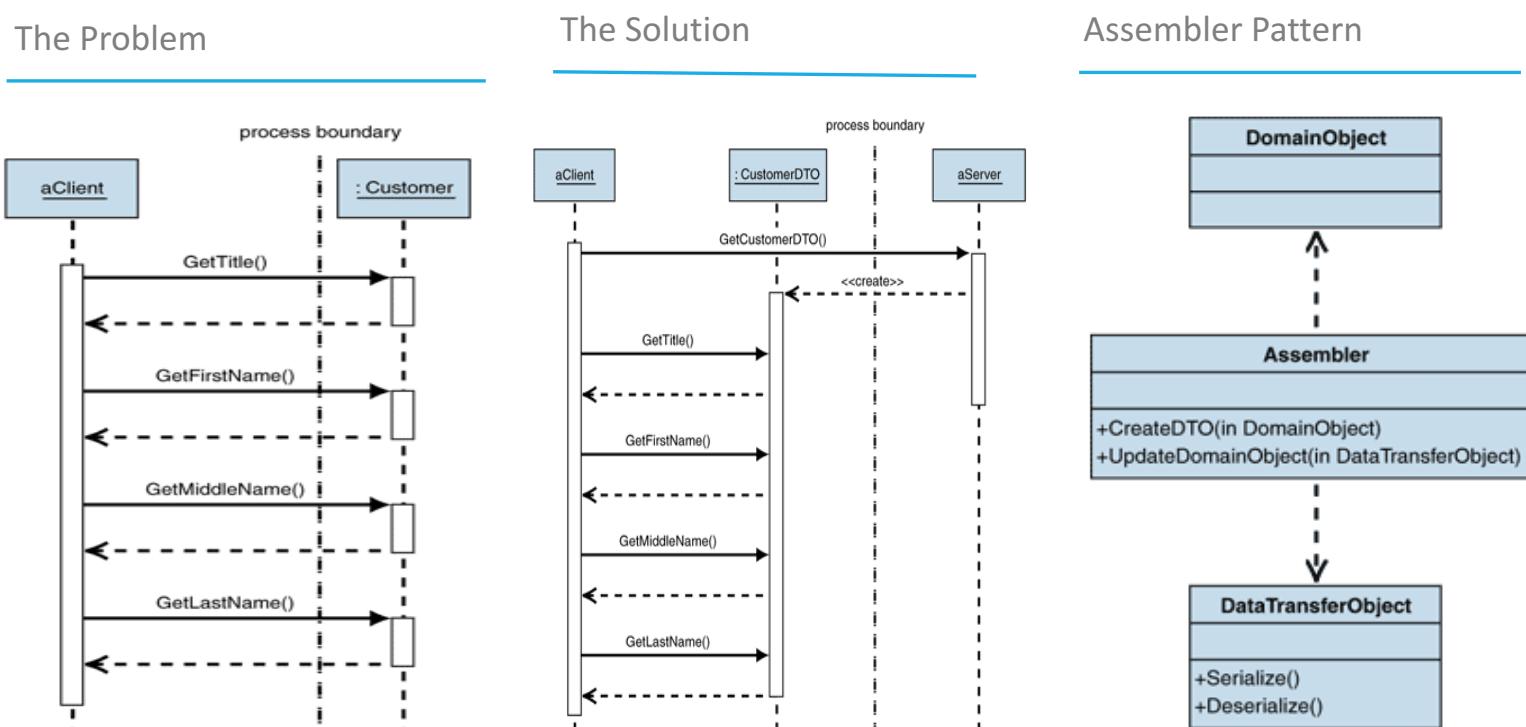
1. Reduced Number of Calls
2. Improved Performance
3. Hidden Internals
4. Discovery of Business objects

## Liabilities

1. Class Explosion
2. Additional Computation
3. Additional Coding Effort

## Security Considerations

Data obtained from untrusted sources, such as user input from a Web page, should be cleansed and validated before being placed into a DTO. Doing so enables you to consider the data in the DTO relatively safe, which simplifies future interactions with the DTO.



# DTO – Data Transfer Object

An object that carries data between processes in order to reduce the number of method calls.

401  
P of EAA

Don't underestimate the cost of [using DTOs].... It's significant, and it's painful - perhaps second only to the cost and pain of object-relational mapping.

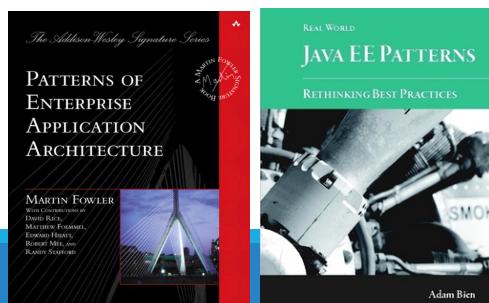
Another argument I've heard is using them in case you want to distribute later. This kind of speculative distribution boundary is what I rail against. Adding remote boundaries adds complexity.

One case where it is useful to use something like a DTO is when you have a significant mismatch between the model in your presentation layer and the underlying domain model.

In this case it makes sense to make presentation specific facade/gateway that maps from the domain model and presents an interface that's convenient for the presentation.

Patterns of Enterprise Application Architecture : Martin Fowler

<http://martinfowler.com/books/eaa.html>



The most misused pattern in the Java Enterprise community is the DTO.

DTO was clearly defined as a solution for a distribution problem.

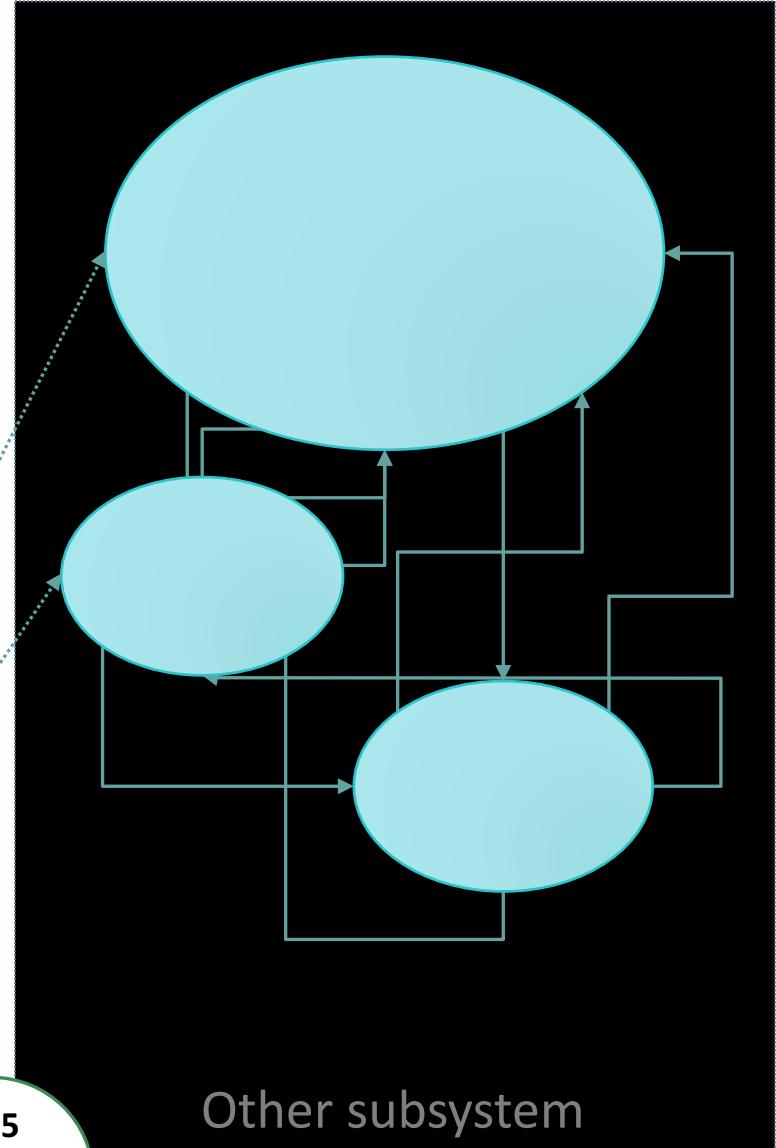
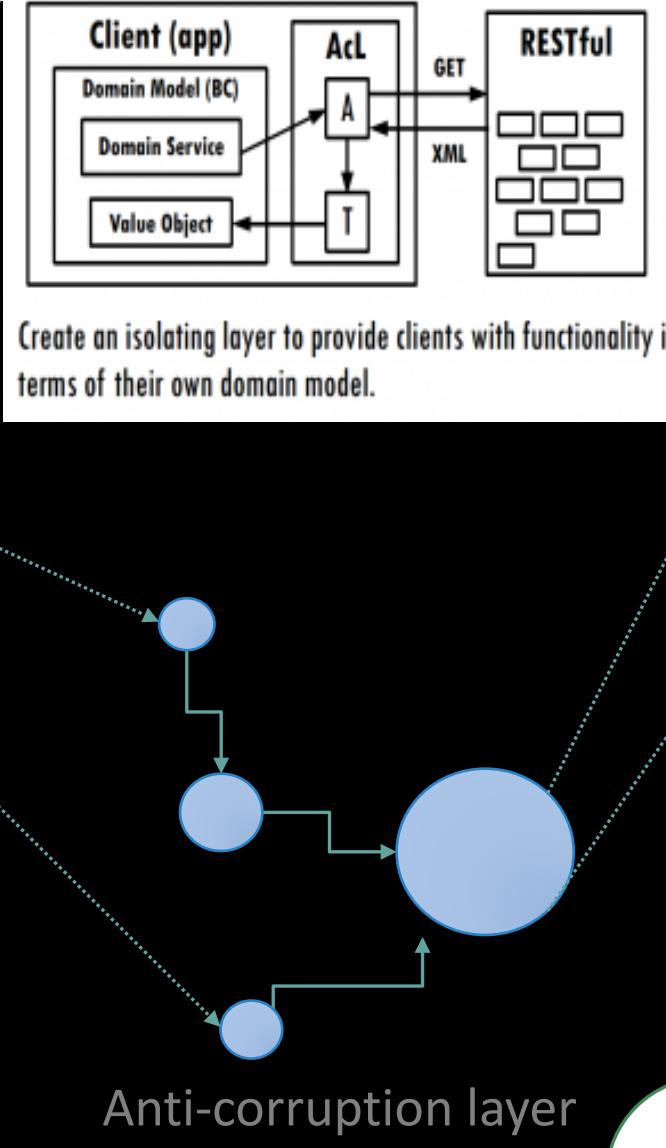
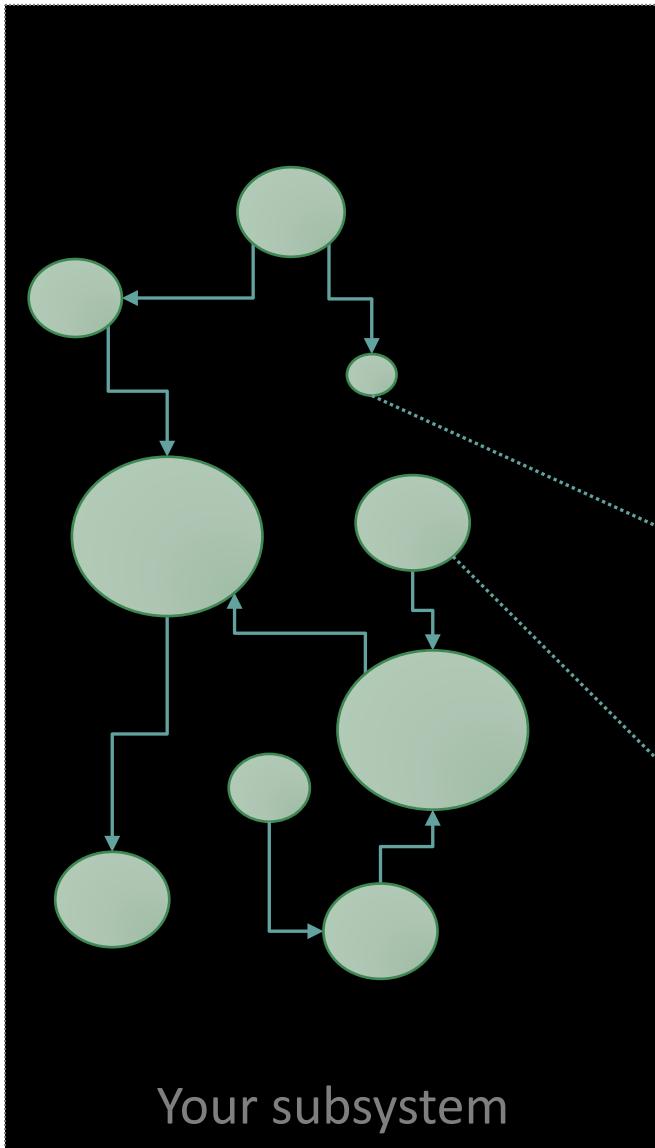
DTO was meant to be a coarse-grained data container which efficiently transports data between processes (tiers).

On the other hand considering a dedicated DTO layer as an investment, rarely pays off and often lead to over engineered bloated architecture.

Real World Java  
EE Patterns  
Adam Bien

<http://realworldpatterns.com>

# Anti Corruption Layer – ACL



# Repository Pattern

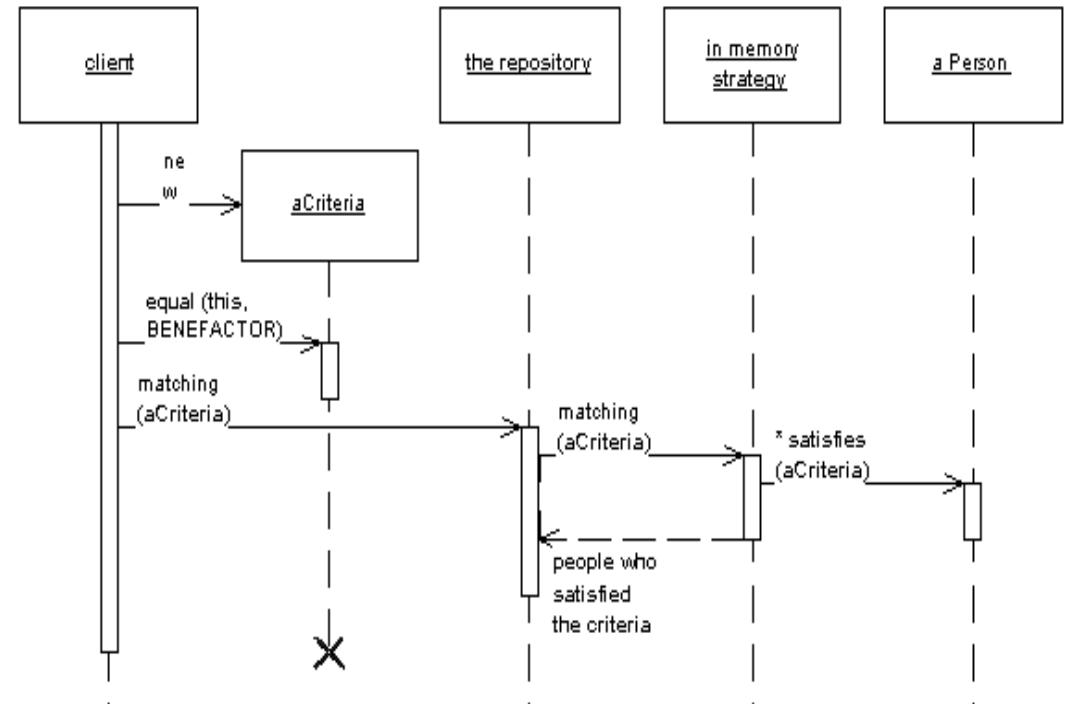
322  
P of EAA

Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.

## Objectives

Use the Repository pattern to achieve one or more of the following objectives:

- You want to maximize the amount of code that can be tested with automation and to isolate the data layer to support unit testing.
- You access the data source from many locations and want to apply centrally managed, consistent access rules and logic.
- You want to implement and centralize a caching strategy for the data source.
- You want to improve the code's maintainability and readability by separating business logic from data or service access logic.
- You want to use business entities that are strongly typed so that you can identify problems at compile time instead of at run time.
- You want to associate a behavior with the related data. For example, you want to calculate fields or enforce complex relationships or business rules between the data elements within an entity.
- You want to apply a domain model to simplify complex business logic.



Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

Repository Pattern Source:

Martin Fowler : <http://martinfowler.com/eaaCatalog/repository.html> | Microsoft : <https://msdn.microsoft.com/en-us/library/ff649690.aspx>

# Anemic Domain Model : Anti Pattern

- There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have.
- The catch comes when you look at the behavior, and you realize that there is **hardly any behavior on these objects**, making them little more than **bags of getters and setters**.
- The fundamental **horror** of this anti-pattern is that **it's so contrary to the basic idea of object-oriented design**; which is to combine data and process together.
- The **anemic domain model** is really just a **procedural style design**, exactly the kind of thing that object bigots like me (and Eric) have been fighting since our early days in Smalltalk.

```

1 package com.fusionfire.examples.commons.utils;
2
3 import java.util.ArrayList;
4
5 public class AnemicUser {
6
7     private String name;
8
9     private boolean isUserLocked;
10
11    private ArrayList<String> addresses;
12
13    public String getName() {
14        return name;
15    }
16
17    public void setName(String name) {
18        this.name = name;
19    }
20
21    public boolean isUserLocked() {
22        return isUserLocked;
23    }
24
25    public void setUserLocked(boolean isUserLocked) {
26        this.isUserLocked = isUserLocked;
27    }
28
29    public ArrayList<String> getAddresses() {
30        return addresses;
31    }
32
33    public void setAddresses(ArrayList<String> addresses) {
34        this.addresses = addresses;
35    }
36
37 }
```

- lockUser()
- unlockUser()
- addAddress(String address)
- removeAddress(String address)



# Procedural Design Vs. Domain Driven Design

```

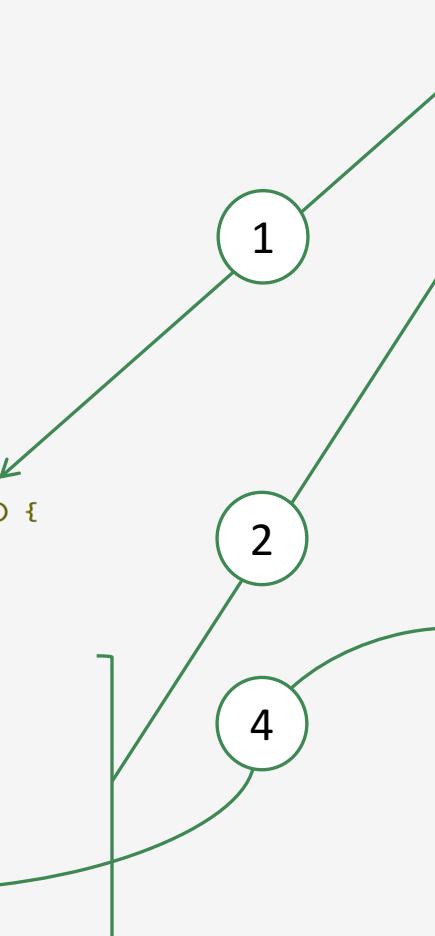
@Stateless
public class ShipmentService {
    public final static int BASIC_COST = 5;

    @PersistenceContext
    private EntityManager em;

    public int getShippingCosts(int loadId) {
        Load load = em.find(Load.class, loadId);
        return computeShippingCost(load);
    }

    int computeShippingCost(Load load){
        int shippingCosts = 0;
        int weight = 0;
        int defaultCost = 0;
        for (OrderItem orderItem : load.getOrderItems()) {
            LoadType loadType = orderItem.getLoadType();
            weight = orderItem.getWeight();
            defaultCost = weight * 5;
            switch (loadType) {
                case BULKY:
                    shippingCosts += (defaultCost + 5);
                    break;
                case LIGHTWEIGHT:
                    shippingCosts += (defaultCost - 1);
                    break;
                case STANDARD:
                    shippingCosts += (defaultCost);
                    break;
                default:
                    throw new IllegalStateException("Unknown type: " + loadType);
            }
        }
        return shippingCosts;
    }
}

```



1. Anemic Entity Structure
2. Massive IF Statements
3. Entire Logic resides in Service Layer
4. Type Dependent calculations are done based on conditional checks in Service Layer

Domain Driven Design with Java EE 6  
By Adam Bien | Javaworld

# Polymorphic Business Logic inside a Domain object

```
@Entity  
public class Load {  
  
    @OneToMany(cascade = CascadeType.ALL)  
    private List<OrderItem> orderItems;  
    @Id  
    private Long id;  
  
    protected Load() {  
        this.orderItems = new ArrayList<OrderItem>();  
    }  
  
    public int getShippingCosts() {  
        int shippingCosts = 0;  
        for (OrderItem orderItem : orderItems) {  
            shippingCosts += orderItem.getShippingCost();  
        }  
        return shippingCosts;  
    }  
    //...  
}
```

Computation of the total cost realized inside a rich Persistent Domain Object (PDO) and not inside a service.

This simplifies creating very complex business rules.

Domain Driven Design with Java EE 6  
By Adam Bien | Javaworld

Source: <http://www.javaworld.com/article/2078042/java-app-dev/domain-driven-design-with-java-ee-6.html>

# Type Specific Computation in a Sub Class

```
@Entity  
public class BulkyItem extends OrderItem{  
  
    public BulkyItem() {  
    }  
  
    public BulkyItem(int weight) {  
        super(weight);  
    }  
  
    @Override  
    public int getShippingCost() {  
        return super.getShippingCost() + 5;  
    }  
}
```

We can change the computation of the shipping cost of a Bulky Item without touching the remaining classes.

It's easy to introduce a new Sub Class without affecting the computation of the total cost in the Load Class.

Domain Driven Design with Java EE 6  
By Adam Bien | Javaworld

# Object Construction : Procedural Way Vs. Builder Pattern

## Procedural Way

```
Load load = new Load();
OrderItem standard = new OrderItem();
standard.setLoadType(LoadType.STANDARD);
standard.setWeight(5);
load.getOrderItems().add(standard);
OrderItem light = new OrderItem();
light.setLoadType(LoadType.LIGHTWEIGHT);
light.setWeight(1);
load.getOrderItems().add(light);
OrderItem bulky = new OrderItem();
bulky.setLoadType(LoadType.BULKY);
bulky.setWeight(1);
load.getOrderItems().add(bulky);
```

## Builder Pattern

```
Load build = new Load.Builder().
    withStandardItem(5).
    withLightweightItem(1).
    withBulkyItem(1).
    build();
```

Domain Driven Design with Java EE 6  
By Adam Bien | Javaworld

# DDD – Summary

1. Ubiquitous Language
2. Aggregate Root
3. Value Object
4. Domain Events
5. Data Transfer Object
6. Repository Pattern
7. Context Map

More on this in Event Sourcing and CQRS Section.



# CRUD / CQRS & Event Sourcing

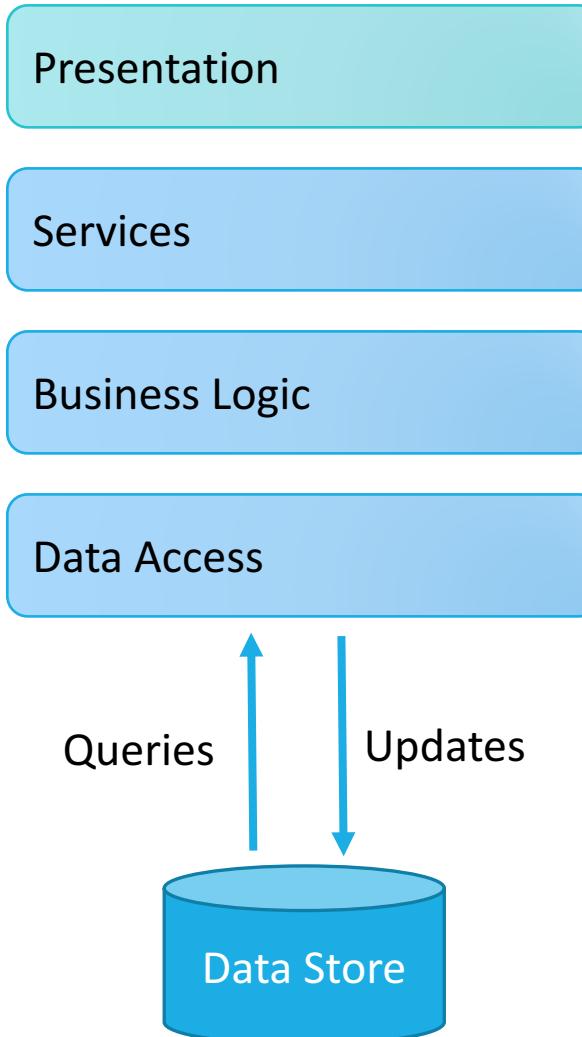
A brief introduction, more in Part 2 of the Series

Event Storming and SAGA

# CRUD and CQRS

Source: <https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs>  
[https://blogs.msdn.microsoft.com/maarten\\_mullender/2004/07/23/crud-only-when-you-can-afford-it-revisite](https://blogs.msdn.microsoft.com/maarten_mullender/2004/07/23/crud-only-when-you-can-afford-it-revisite)

## Traditional CRUD Architecture

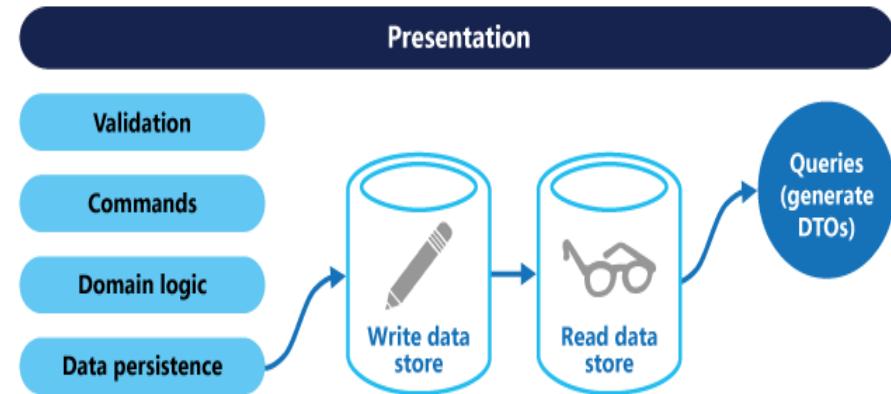
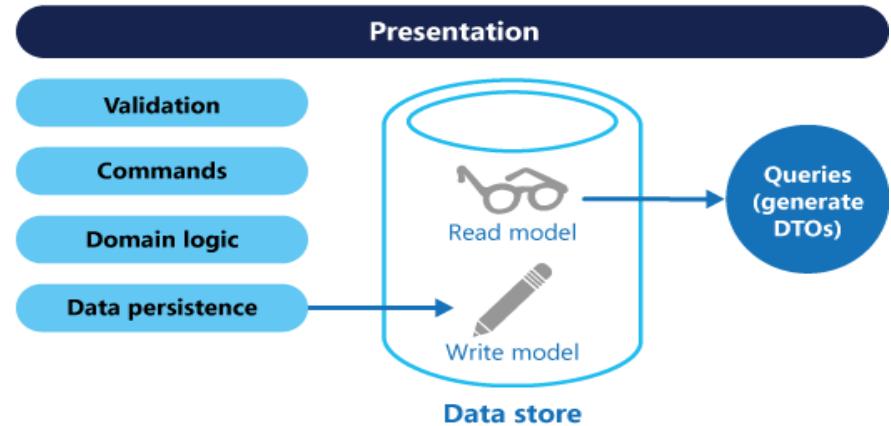


## CRUD Disadvantages

- **A mismatch between the read and write representations of the data.**
- **It risks data contention when records are locked in the data store** in a collaborative domain, where multiple actors operate in parallel on the same set of data. These risks increase as the complexity and throughput of the system grows.
- **It can make managing security and permissions more complex** because each entity is subject to both read and write operations, which might expose data in the wrong context.

# Event Sourcing & CQRS (Command and Query Responsibility Segregation)

- In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository.
- CQRS is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces.
- CQRS should only be used on specific portions of a system in Bounded Context (in DDD).
- CQRS should be used along with Event Sourcing.



Java Axon Framework Resource : <http://www.axonframework.org>

MSDN – Microsoft <https://msdn.microsoft.com/en-us/library/dn568103.aspx> |  
Martin Fowler : CQRS – <http://martinfowler.com/bliki/CQRS.html>



Axon  
Framework  
For Java

CQS :  
Bertrand Meyer

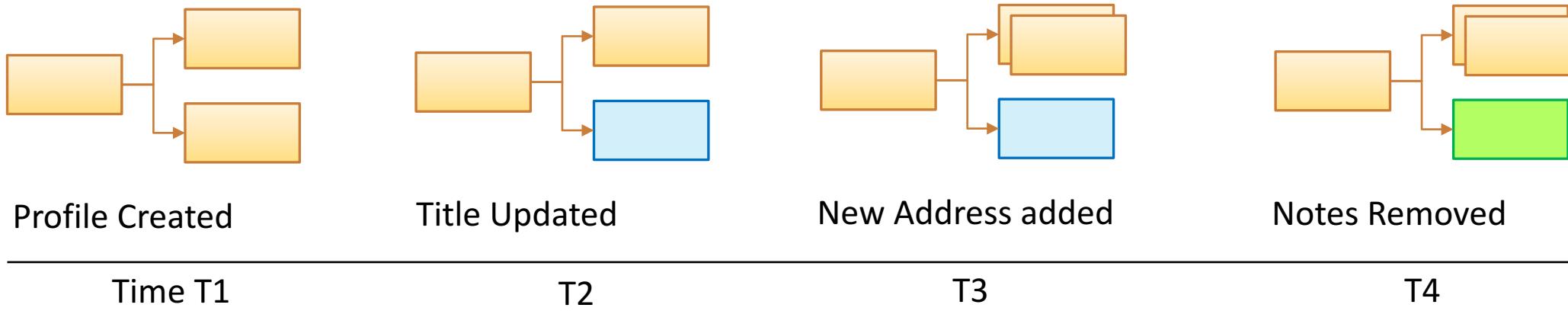


Greg  
Young



# Event Sourcing Intro

## Standard CRUD Operations – Customer Profile – Aggregate Root



## Event Sourcing and Derived Aggregate Root

### Commands

1. Create Profile
2. Update Title
3. Add Address
4. Delete Notes

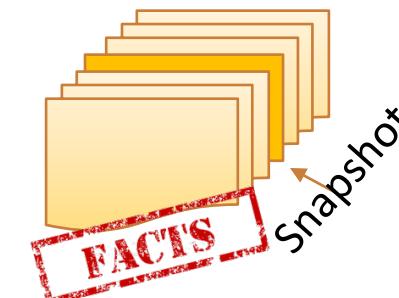
### Events

1. Profile Created Event
2. Title Updated Event
3. Address Added Event
4. Notes Deleted Event

2

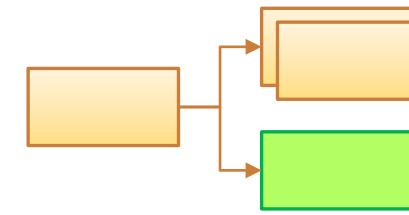
3

### Event store



Single Source of Truth

Derived



Current State of the  
Customer Profile

4

Greg Young



# Event Sourcing and CQRS Design Example

## Domain

The example focus on a concept of a Café which tracks the visit of an individual or group to the café. When people arrive at the café and take a table, a **tab** is opened. They may then **order** drinks and food. **Drinks are served** immediately by the table staff, however **food** must be cooked by a chef. Once the chef **prepared** the food it can then be **served**.

Events	Commands	Aggregates
<ul style="list-style-type: none"> <li>• TabOpened</li> <li>• DrinksOrdered</li> <li>• FoodOrdered</li> <li>• DrinksCancelled</li> <li>• FoodCancelled</li> <li>• DrinksServed</li> <li>• FoodPrepared</li> <li>• FoodServed</li> <li>• TabClosed</li> </ul> <p>An Event Stream which is an <b>immutable</b> collection of events up until a specific version of an <b>aggregate</b>. The purpose of the version is to implement optimistic locking:</p>	<ul style="list-style-type: none"> <li>• OpenTab</li> <li>• PlaceOrder</li> <li>• AmendOrder</li> <li>• MarkDrinksServed</li> <li>• MarkFoodPrepared</li> <li>• MarkFoodServed</li> <li>• CloseTab</li> </ul> <p>Commands are things that indicate <b>requests</b> to our domain. While an event states that something certainly happened, a command may be <b>accepted</b> or <b>rejected</b>. An accepted command leads to zero or more events being emitted to incorporate new facts into the system. A rejected command leads to some kind of exception.</p>	<ul style="list-style-type: none"> <li>• A Single Object, which doesn't reference any others.</li> <li>• An isolated Graph of objects, with One object designated as the Root of the Aggregate.</li> </ul>

## Exception

- CannotCancelServedItem
- TabHasUnservedItem
- MustPayEnough

An important part of the modeling process is thinking about the things that can cause a command to be refused.

# Event Storming : Restaurant Dining Example – Customer Journey

Processes



1

When people arrive at the Restaurant and take a table, a **Table** is **opened**. They may then **order drinks** and **food**. **Drinks** are **served** immediately by the table staff, however **food** must be **cooked** by a **chef**. Once the **chef prepared** the food it can then be **served**. **Table** is **closed** when the **bill** is prepared.

Commands

- Add Drinks
- Add Food
- Update Food

2

Customer Journey thru Dinning Processes

- **Open Table**
- Remove Soda
- Add Juice
- Add Food 1
- Add Soda
- Add Food 2
- Add Appetizer 1
- Place Order
- Add Appetizer 2
- **Close Table**

ES Aggregate

- Dinning Order
- Billable Order

4

Food Menu



Dining



Kitchen



Order



Payment



Microservices

Events

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued

3

- Table Opened
- Remove Soda
- Juice Added
- Food 1 Added
- Soda Added
- Food 2 Added
- Appetizer 1 Added
- Appetizer Served
- Order Placed
- Table Closed

- Bill Prepared
- Payment Processed

- Payment Approved
- Payment Declined
- Cash Paid

# Summary – Event Sourcing and CQRS

---

1. Immutable Events
2. Events represents the state change in Aggregate Root
3. Aggregates are Derived from a Collection of Events.
4. Separate Read and Write Models
5. Commands (originated from user or systems) creates Events.
6. Commands and Queries are always separated and possibly reads and writes using different data models.

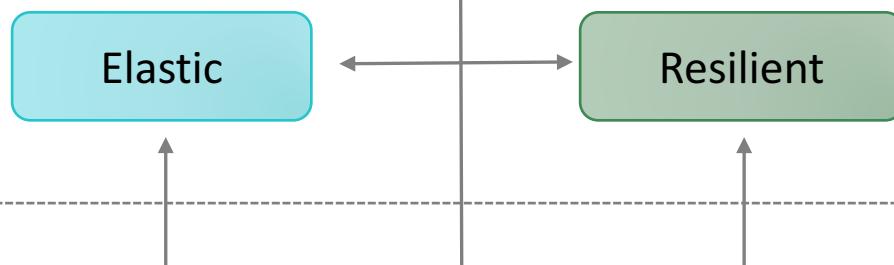
# Functional Reactive Programming

# Functional Reactive Programming

Value



Means



Form



Principles		What it means?
Responsive	thus	React to users demand
Resilient	thus	React to errors and failures
Elastic	thus	React to load
Message-Driven	thus	React to events and messages

1. A **responsive, maintainable & Extensible** application is the goal.
2. A **responsive** application is both **scalable (Elastic)** and **resilient**.
3. Responsiveness is impossible to achieve without both scalability and resilience.
4. A **Message-Driven** architecture is the foundation of scalable, resilient, and ultimately responsive systems.



**Reactive  
Extensions  
(Rx)**

Source: <http://reactivex.io/>

# 4 Building Blocks of RxJava



1

**Observable**

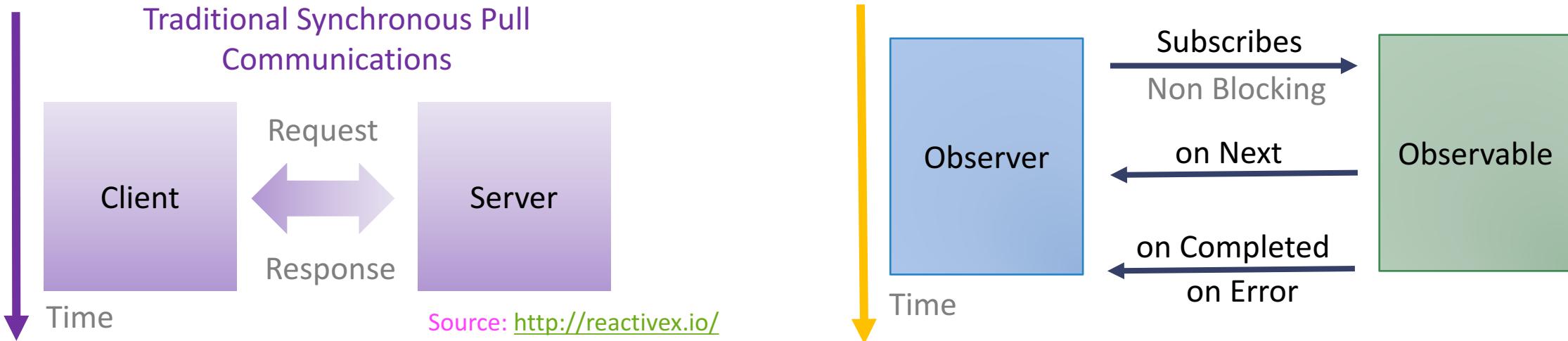
Source of Data Stream [ Sender ]

2

**Observer**

Listens for emitted values [ Receiver ]

1. The **Observer** subscribes (listens) to the **Observable**
2. **Observer** react to what ever item or sequence of items the **Observable** emits.
3. Many **Observers** can subscribe to the same **Observable**



# 4 Building Blocks of RxJava



3

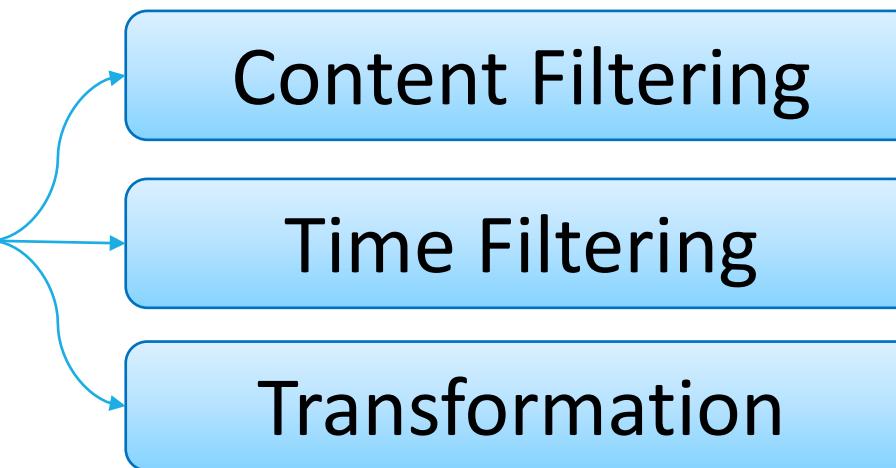
## Schedulers

**Schedulers** are used to manage and control concurrency.

1. `observeOn`: Thread Observable is executed
2. `subscribeOn`: Thread subscribe is executed

4

## Operators

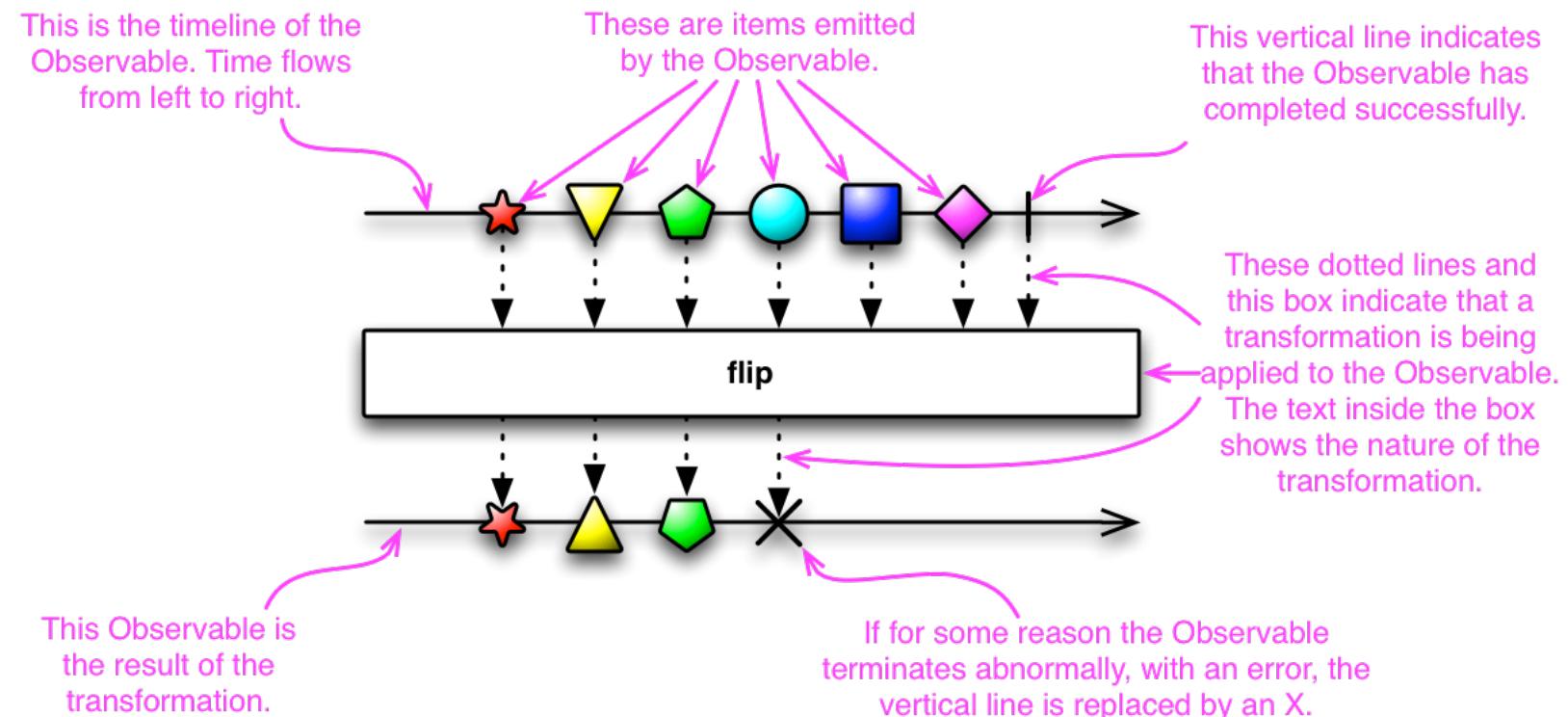


**Operators** that let you **Transform**, **Combine**, **Manipulate**, and work with the sequence of items emitted by **Observables**

Source: <http://reactivex.io/>

# Observable Design Pattern (Marble Diagram)

- An *Observer* subscribes to an *Observable*.
- Then that **observer reacts** to whatever item or sequence of items the *Observable emits*.



Source: <http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html> | <http://rxmarbles.com>

- Allows for Concurrent Operations: the observer does not need to block while waiting for the observable to emit values
- Observer waits to receive values when the observable is ready to emit them
- Based on push rather than pull

Source: <http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html>



1. The ability for the producer **to signal to the consumer that there is no more data available** (a foreach loop on an Iterable completes and returns normally in such a case; an Observable calls its observer's onComplete method)
2. The ability for the producer **to signal to the consumer that an error has occurred** (an Iterable throws an exception if an error takes place during iteration; an Observable calls its observer's onError method)
3. Multiple Thread Implementations and hiding those details.
4. Dozens of Operators to handle data.



# Compare Iterable Vs. Observable

Building Block

1



Observable is the asynchronous / push dual to the synchronous pull Iterable

Observables are:

- **Composable:** Easily chained together or combined
- **Flexible:** Can be used to emit:
  - A scalar value (network result)
  - Sequence (items in a list)
  - **Infinite streams** (weather sensor)
- **Free from callback hell:** Easy to transform one asynchronous stream into another

Event	Iterable (Pull)	Observable (Push)
Retrieve Data	T next()	onNext(T)
Discover Error	throws Exception	onError (Exception)
Complete	!hasNext()	onComplete()

Source: <http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html>

# Comparison : Iterable / Streams / Observable



## Java 6 – Blocking Call

```
/**
 * Iterable Serial Operations Example
 * Java 6 & 7
 */
public void testIterable(AppleBasket _basket) {
    Iterable<Apple> basket = _basket.iterator();
    FruitProcessor<Apple> fp =
        new FruitProcessor<Apple>("IT");
    try {
        // Serial Operations
        for(Apple apple : basket) {
            fp.onNext(apple);
        }
        fp.onCompleted();
    } catch (Exception e) {
        fp.onError(e);
    }
}
```

First Class Visitor (Consumer)  
Serial Operations



## Java 8 – Blocking Call

```
/**
 * Parallel Streams Example
 * Java 8 with Lambda Expressions
 */
public void testParallelStream(AppleBasket _basket) {
    Collection<Apple> basket = _basket.collection();
    FruitProcessor<Apple> fp =
        new FruitProcessor<Apple>("PS");
    try {
        // Parallel Operations
        basket
            .parallelStream()
            .forEach(apple -> fp.onNext(apple));
        fp.onCompleted();
    } catch (Exception e) {
        fp.onError(e);
    }
}
```

Parallel Streams (10x Speed)  
Still On Next, On Complete and  
On Error are Serial Operations



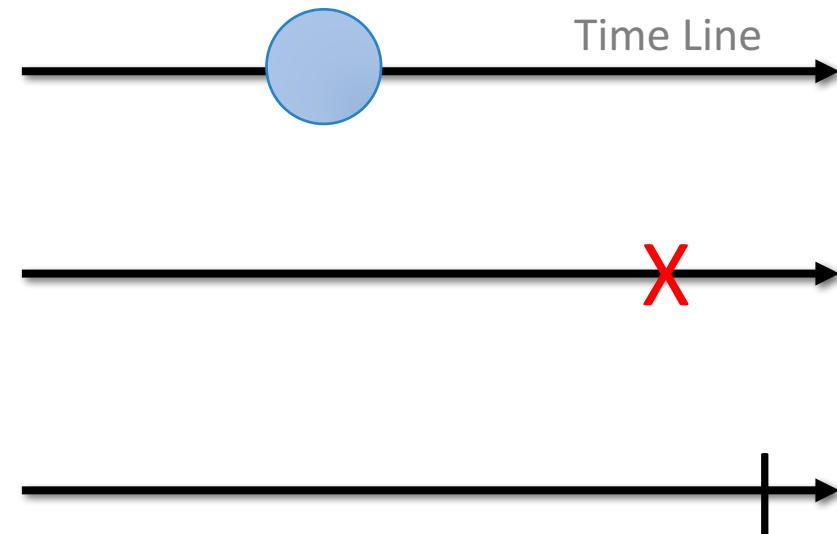
## Rx Java - Freedom

```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx Java
 */
public void testObservable1() {
    Observable<Apple> basket = fruitBasketObservable();
    Observer<Apple> fp = fruitProcessor("01");
    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            apple -> fp.onNext(apple),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );
}
```

Completely Asynchronous  
Operations

## Methods:

- **onNext(T)**
- **onError(Throwable T)**
- **onComplete()**



onError / onComplete called exactly once

Source: <http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html>



- If you want to introduce **multithreading** into your cascade of **Observable operators**, you can do so by instructing those operators (or particular Observables) to operate on particular *Schedulers*.
- By default, an **Observable** and the chain of **operators** that you apply to it will do its work, and will notify its **observers**, on the same thread on which its **Subscribe** method is called.
- The **SubscribeOn** operator changes this behavior by specifying a different **Scheduler** on which the **Observable** should operate. The **ObserveOn** operator specifies a different **Scheduler** that the **Observable** will use to send notifications to its **observers**.

Source: <http://reactivex.io/documentation/scheduler.html>



# Functional Reactive Programming : Design Patterns

## Single Component Pattern

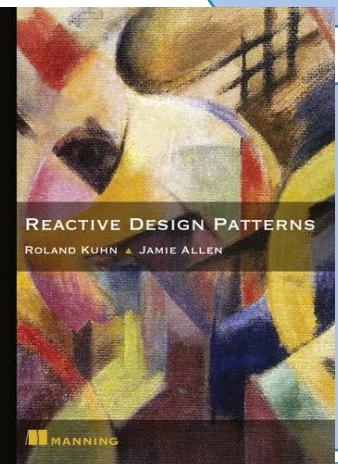
A Component shall do ONLY one thing,  
But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

## Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)  
Popularized by Netflix Chaos Monkey. Erlang Philosophy



## Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helling: Life Beyond Distributed Transactions CIDR 2007

# Summary – Functional Reactive Programming

1. A ***responsive, maintainable & Extensible*** application is the goal.
2. A ***responsive*** application is both ***scalable (Elastic)*** and ***resilient***.
3. Responsiveness is impossible to achieve without both scalability and resilience.
4. A ***Message-Driven*** architecture is the foundation of scalable, resilient, and ultimately responsive systems.

# Scalability

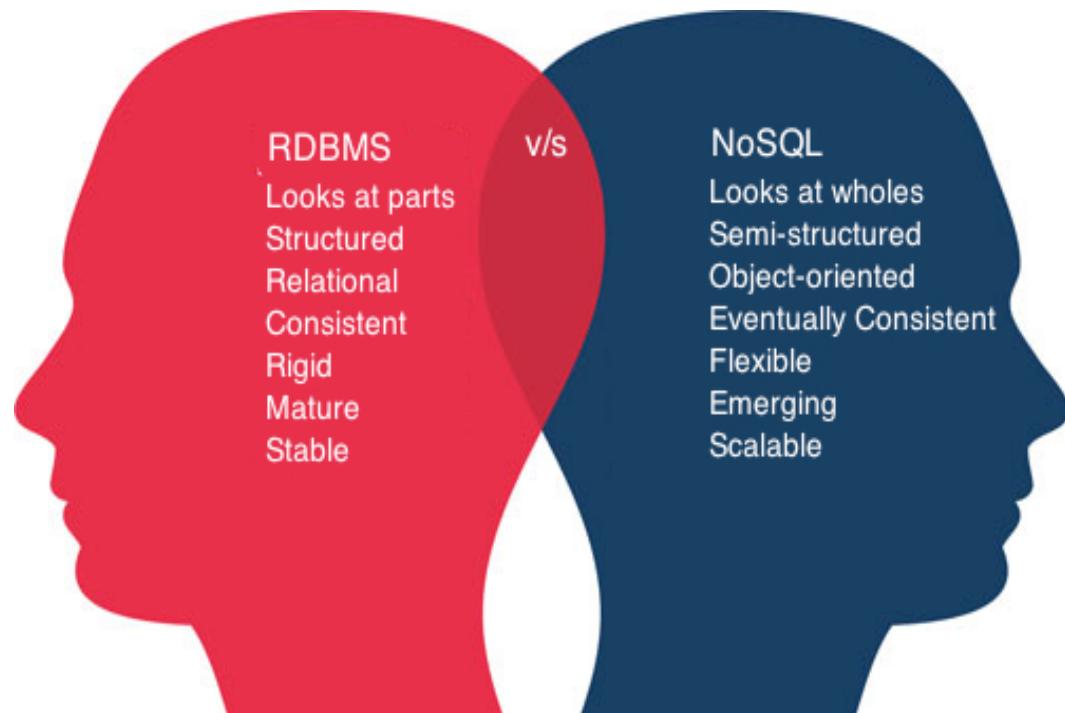
- CAP Theorem
- Distributed Transactions : 2 Phase Commit
- SAGA Design Pattern
- Scalability Lessons from EBay
- Design Patterns
- References

# CAP Theorem

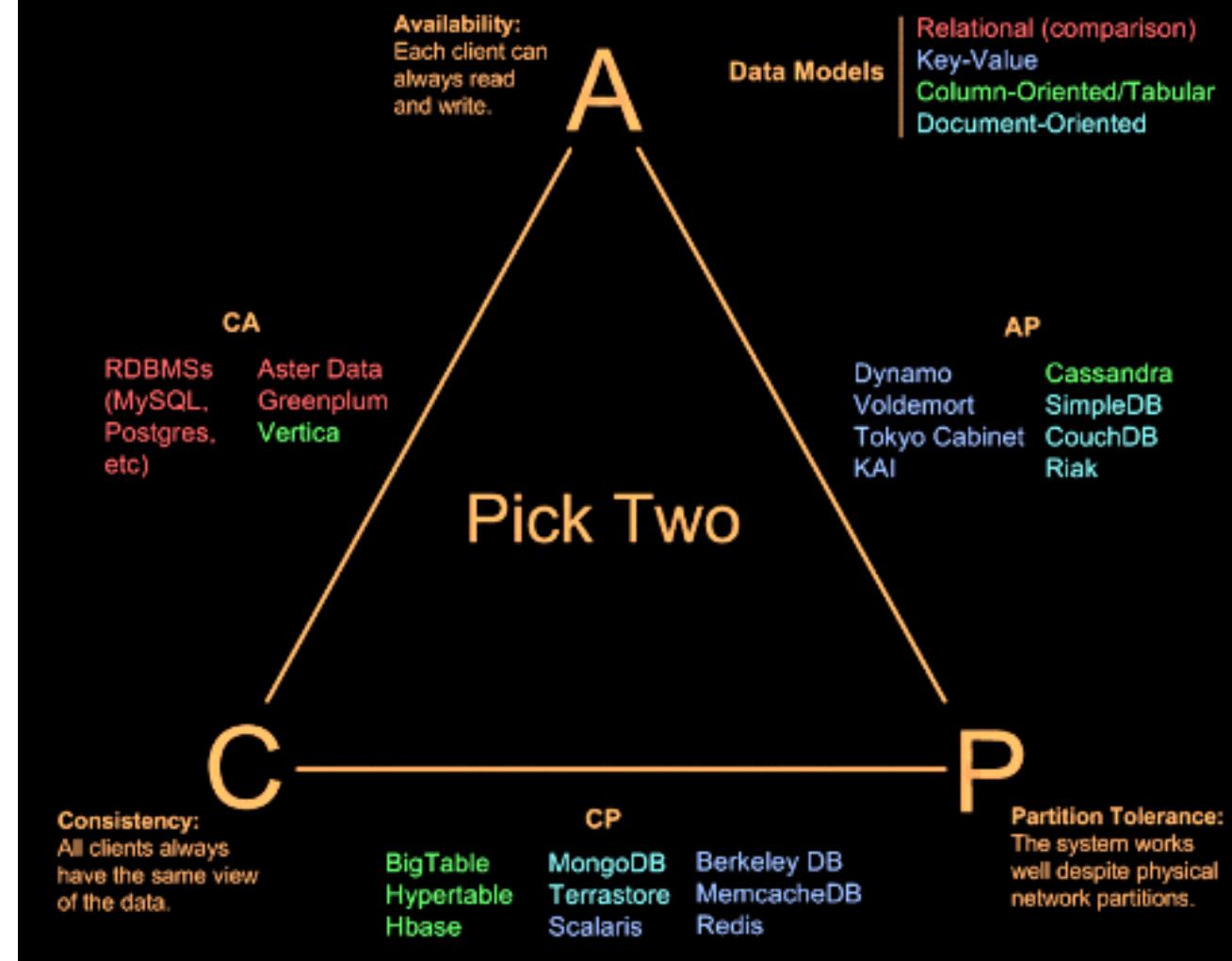
by Eric Allen Brewer

Pick Any 2!!

Say NO to 2 Phase Commit ☺



# Visual Guide to NoSQL Systems



*"In a network subject to communication failures, it is impossible for any web service to implement an atomic read / write shared memory that guarantees a response to every request."*

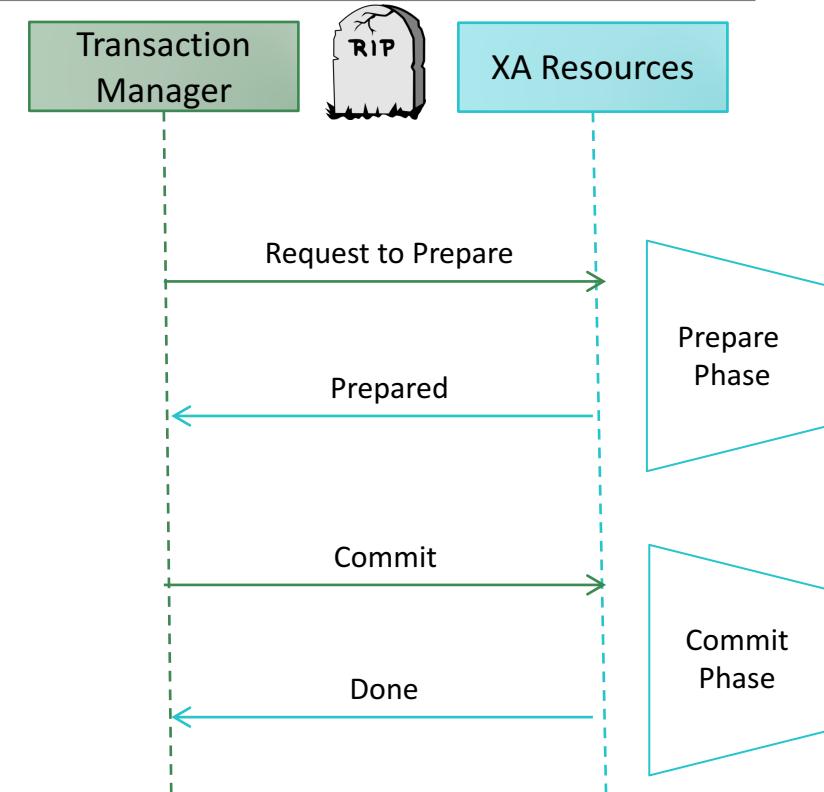
Source: [http://en.wikipedia.org/wiki/Eric\\_Brewer\\_\(scientist\)](http://en.wikipedia.org/wiki/Eric_Brewer_(scientist))

# Distributed Transactions : 2 Phase Commit

2 PC or not 2 PC, Wherefore Art Thou XA?

## How does 2PC impact scalability?

- Transactions are committed in two phases.
- This involves communicating with every database (XA Resources) involved to determine if the transaction will commit in the first phase.
- During the second phase each database is asked to complete the commit.
- While all of this coordination is going on, locks in all of the data sources are being held.
- ***The longer duration locks create the risk of higher contention.***
- ***Additionally, the two phases require more database processing time than a single phase commit.***
- ***The result is lower overall TPS in the system.***



## Solution : Resilient System

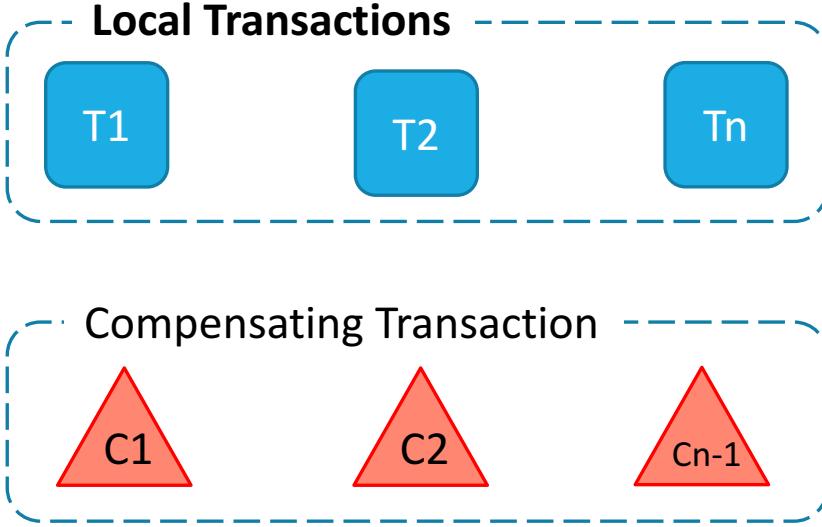
- Event Based
- Design for failure
- Asynchronous Recovery
- Make all operations idempotent.
- Each DB operation is a 1 PC

Source : Pat Helland (Amazon) : Life Beyond Distributed Transactions Distributed Computing : <http://dances.github.io/Pages/>

# SAGA Design Pattern instead of 2PC

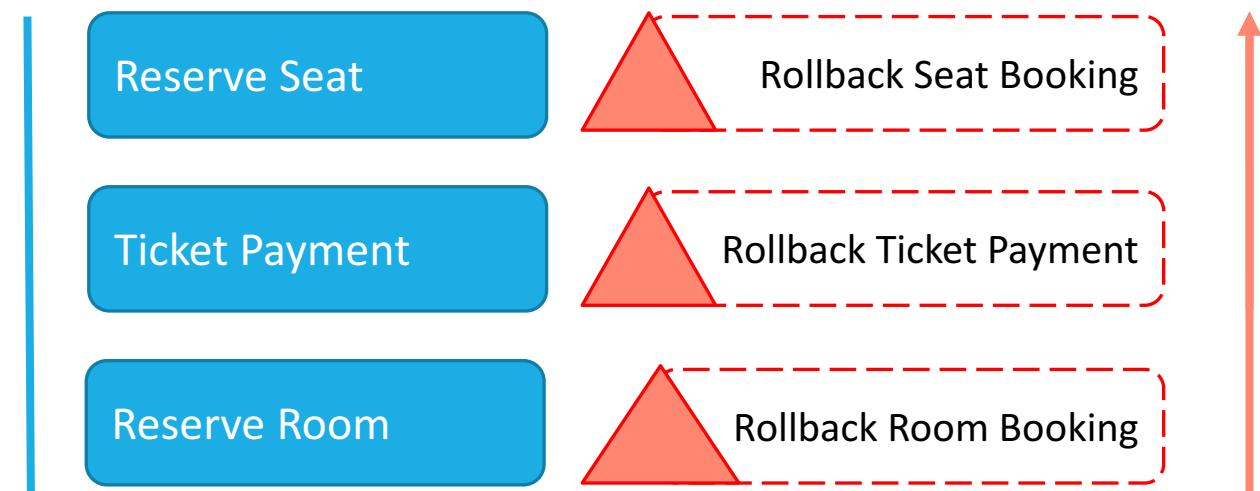
**Long Lived Transactions (LLTs)** hold on to DB resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions.

**Divide long-lived, distributed transactions into quick local ones with compensating actions for recovery.**



Source: [SAGAS \(1987\) Hector Garcia Molina](#) / Kenneth Salem, Dept. of Computer Science, Princeton University, NJ, USA

## Flight Ticket & Hotel Booking Example



**BASE (Basic Availability, Soft State, Eventual Consistency)**

# Scalability Best Practices : Lessons from ebay

16 April 2018

Best Practices	Highlights
#1 Partition By Function	<ul style="list-style-type: none"> <li>Decouple the Unrelated Functionalities.</li> <li>Selling functionality is served by one set of applications, bidding by another, search by yet another.</li> <li>16,000 App Servers in 220 different pools</li> <li>1000 logical databases, 400 physical hosts</li> </ul>
#2 Split Horizontally	<ul style="list-style-type: none"> <li>Break the workload into manageable units.</li> <li>eBay's interactions are stateless by design</li> <li>All App Servers are treated equal and none retains any transactional state</li> <li>Data Partitioning based on specific requirements</li> </ul>
#3 Avoid Distributed Transactions	<ul style="list-style-type: none"> <li>2 Phase Commit is a pessimistic approach comes with a big COST</li> <li>CAP Theorem (Consistency, Availability, Partition Tolerance). Apply any two at any point in time.</li> <li>@ eBay No Distributed Transactions of any kind and NO 2 Phase Commit.</li> </ul>
#4 Decouple Functions Asynchronously	<ul style="list-style-type: none"> <li>If Component A calls component B synchronously, then they are tightly coupled. For such systems to scale A you need to scale B also.</li> <li>If Asynchronous A can move forward irrespective of the state of B</li> <li>SEDA (Staged Event Driven Architecture)</li> </ul>
#5 Move Processing to Asynchronous Flow	<ul style="list-style-type: none"> <li>Move as much processing towards Asynchronous side</li> <li>Anything that can wait should wait</li> </ul>
#6 Virtualize at All Levels	<ul style="list-style-type: none"> <li>Virtualize everything. eBay created their on O/R layer for abstraction</li> </ul>
#7 Cache Appropriately	<ul style="list-style-type: none"> <li>Cache Slow changing, read-mostly data, meta data, configuration and static data.</li> </ul>

Source: <http://www.infoq.com/articles/ebay-scalability-best-practices>

# Summary

---

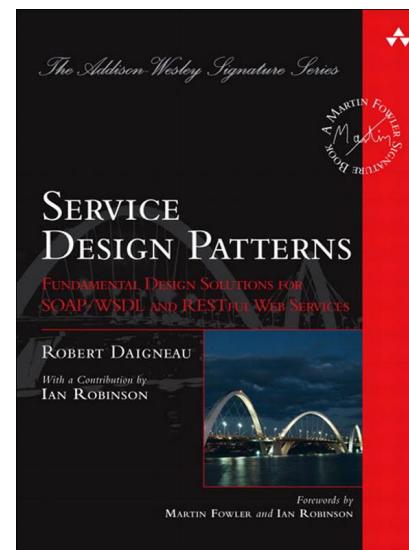
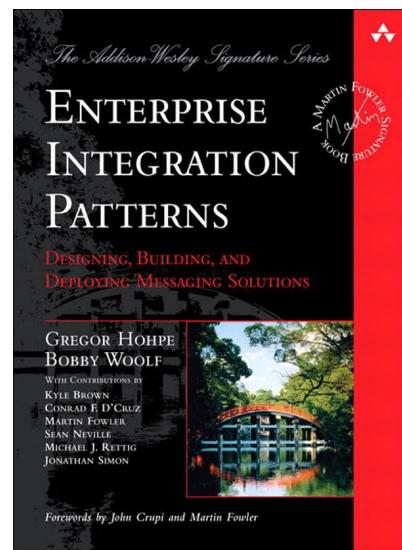
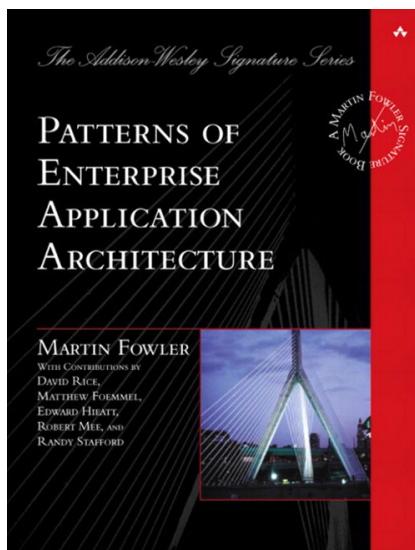
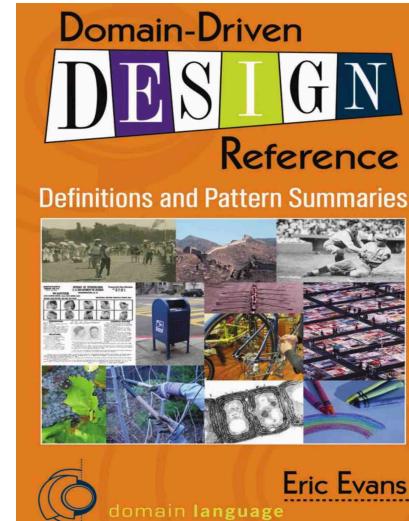
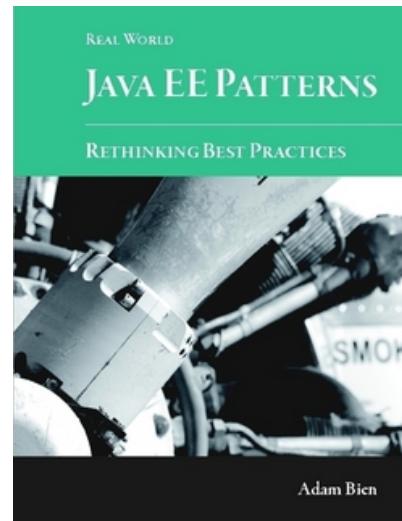
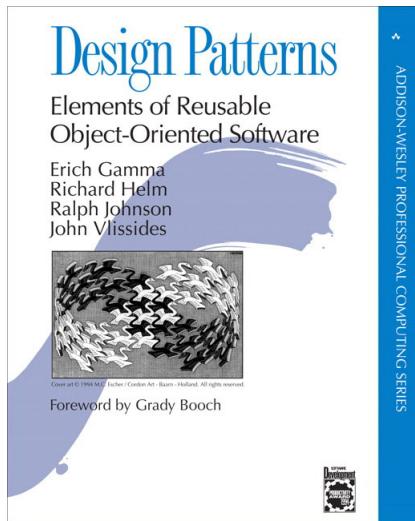
1. Highly Scalable & Resilient Architecture
2. Technology Agnostic
3. Easy to Deploy
4. SAGA for Distributed Transaction
5. Faster Go To Market

In a Micro Service Architecture,

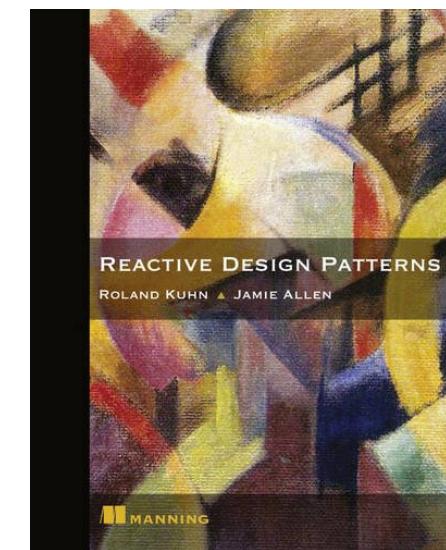
The **services tend to get simpler**, but the **architecture tends to get more complex**.

That **complexity** is often managed with **Tooling, Automation, and Process**.

# Design Patterns



**Design Patterns** are solutions to general problems that software developers faced during software development.



# References

---

1. Lewis, James, and Martin Fowler. "[Microservices: A Definition of This New Architectural Term](#)", March 25, 2014.
2. Miller, Matt. "[Innovate or Die: The Rise of Microservices](#)". *e Wall Street Journal*, October 5, 2015.
3. Newman, Sam. [\*Building Microservices\*](#). O'Reilly Media, 2015.
4. Alagarasan, Vijay. "[Seven Microservices Anti-patterns](#)", August 24, 2015.
5. Cockcroft, Adrian. "[State of the Art in Microservices](#)", December 4, 2014.
6. Fowler, Martin. "[Microservice Prerequisites](#)", August 28, 2014.
7. Fowler, Martin. "[Microservice Tradeoffs](#)", July 1, 2015.
8. Humble, Jez. "[Four Principles of Low-Risk Software Release](#)", February 16, 2012.
9. [Zuul Edge Server](#), Ketan Gote, May 22, 2017
10. [Ribbon, Hystrix using Spring Feign](#), Ketan Gote, May 22, 2017
11. [Eureka Server with Spring Cloud](#), Ketan Gote, May 22, 2017
12. [Apache Kafka, A Distributed Streaming Platform](#), Ketan Gote, May 20, 2017
13. [Functional Reactive Programming](#), Araf Karsh Hamid, August 7, 2016
14. [Enterprise Software Architectures](#), Araf Karsh Hamid, July 30, 2016
15. [Docker and Linux Containers](#), Araf Karsh Hamid, April 28, 2015

# References

## Domain Driven Design

16. Oct 27, 2012 [What I have learned about DDD Since the book](#). By Eric Evans
17. Mar 19, 2013 [Domain Driven Design](#) By Eric Evans
18. May 16, 2015 Microsoft Ignite: [Domain Driven Design for the Database Driven Mind](#)
19. Jun 02, 2015 [Applied DDD in Java EE 7 and Open Source World](#)
20. Aug 23, 2016 [Domain Driven Design the Good Parts](#) By Jimmy Bogard
21. Sep 22, 2016 GOTO 2015 – [DDD & REST Domain Driven API's for the Web](#). By Oliver Gierke
22. Jan 24, 2017 Spring Developer – [Developing Micro Services with Aggregates](#). By Chris Richardson
23. May 17. 2017 DEVOXX – [The Art of Discovering Bounded Contexts](#). By Nick Tune

## Event Sourcing and CQRS

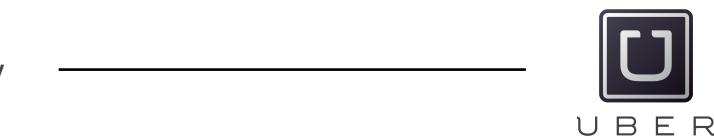
23. Nov 13, 2014 GOTO 2014 – [Event Sourcing](#). By Greg Young
24. Mar 22, 2016 Spring Developer – [Building Micro Services with Event Sourcing and CQRS](#)
25. Apr 15, 2016 YOW! Nights – [Event Sourcing](#). By Martin Fowler
26. May 08, 2017 [When Micro Services Meet Event Sourcing](#). By Vinicius Gomes

# References

- 
27. MSDN – Microsoft <https://msdn.microsoft.com/en-us/library/dn568103.aspx>
  28. Martin Fowler : CQRS – <http://martinfowler.com/bliki/CQRS.html>
  29. Udi Dahan : CQRS – <http://www.udidahan.com/2009/12/09/clarified-cqrs/>
  30. Greg Young : CQRS - <https://www.youtube.com/watch?v=JHGkaShoyNs>
  31. Bertrand Meyer – CQS - [http://en.wikipedia.org/wiki/Bertrand\\_Meyer](http://en.wikipedia.org/wiki/Bertrand_Meyer)
  32. CQS : [http://en.wikipedia.org/wiki/Command–query\\_separation](http://en.wikipedia.org/wiki/Command–query_separation)
  33. CAP Theorem : [http://en.wikipedia.org/wiki/CAP\\_theorem](http://en.wikipedia.org/wiki/CAP_theorem)
  34. CAP Theorem : <http://www.julianbrowne.com/article/viewer/brewers-cap-theorem>
  35. CAP [12 years how the rules have changed](#)
  36. eBay Scalability Best Practices : <http://www.infoq.com/articles/ebay-scalability-best-practices>
  37. Pat Helland (Amazon) : [Life beyond distributed transactions](#)
  38. Stanford University: Rx <https://www.youtube.com/watch?v=y9xudo3C1Cw>
  39. Princeton University: [SAGAS \(1987\) Hector Garcia Molina / Kenneth Salem](#)
  40. Rx Observable : <https://dzone.com/articles/using-rx-java-observable>

# References – Micro Services – Videos

- 41. Martin Fowler – Micro Services : <https://www.youtube.com/watch?v=2yko4TbC8cl&feature=youtu.be&t=15m53s>
- 42. GOTO 2016 – [Microservices at NetFlix Scale](#): Principles, Tradeoffs & Lessons Learned. By R Meshenberg
- 43. Mastering Chaos – [A NetFlix Guide to Microservices](#). By Josh Evans
- 44. GOTO 2015 – [Challenges Implementing Micro Services](#) By Fred George
- 45. GOTO 2016 – [From Monolith to Microservices at Zalando](#). By Rodrigue Scafer
- 46. GOTO 2015 – [Microservices @ Spotify](#). By Kevin Goldsmith
- 47. Modelling Microservices @ Spotify : <https://www.youtube.com/watch?v=7XDA044tl8k>
- 48. GOTO 2015 – [DDD & Microservices: At last, Some Boundaries](#) By Eric Evans
- 49. GOTO 2016 – [What I wish I had known before Scaling Uber to 1000 Services](#). By Matt Ranney
- 50. DDD Europe – [Tackling Complexity in the Heart of Software](#) By Eric Evans, April 11, 2016
- 51. AWS re:Invent 2016 – [From Monolithic to Microservices: Evolving Architecture Patterns](#). By Emerson L, Gilt D. Chiles
- 52. AWS 2017 – [An overview of designing Microservices based Applications on AWS](#). By Peter Dalbhanjan
- 53. GOTO Jun, 2017 – [Effective Microservices in a Data Centric World](#). By Randy Shoup.
- 54. GOTO July, 2017 – [The Seven \(more\) Deadly Sins of Microservices](#). By Daniel Bryant
- 55. Sept, 2017 – [Airbnb, From Monolith to Microservices: How to scale your Architecture](#). By Melanie Cubula
- 56. GOTO Sept, 2017 – [Rethinking Microservices with Stateful Streams](#). By Ben Stopford.
- 57. GOTO 2017 – [Microservices without Servers](#). By Glynn Bird.



# Thank you

Araf Karsh Hamid : Co-Founder / CTO  
[araf.karsh@metamagic.in](mailto:araf.karsh@metamagic.in)

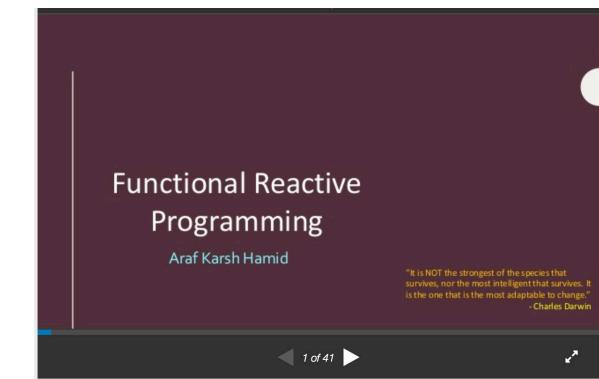
USA: +1 (973) 969-2921

India: +91.999.545.8627

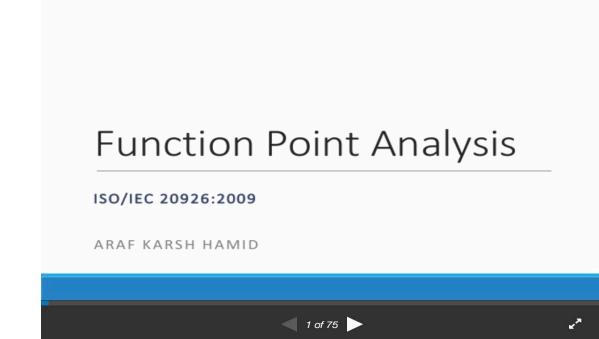
Skype / LinkedIn / Twitter / Slideshare : arafkarsh  
<http://www.slideshare.net/arafkarsh>  
<https://www.linkedin.com/in/arafkarsh/>



<http://www.slideshare.net/arafkarsh/software-architecture-styles-64537120>



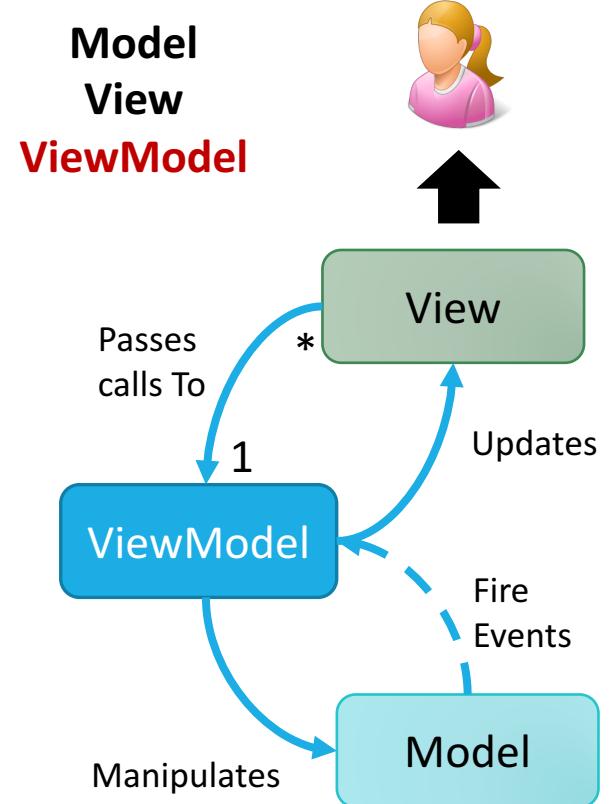
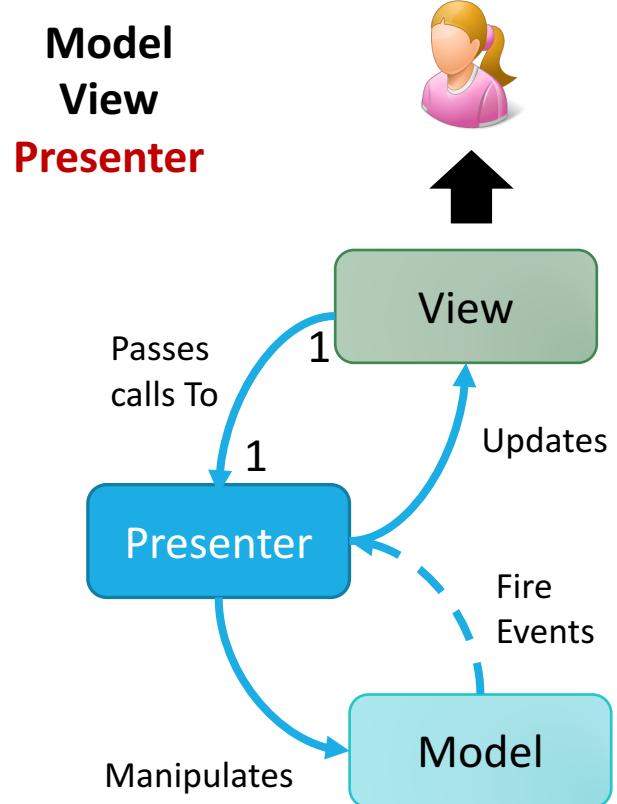
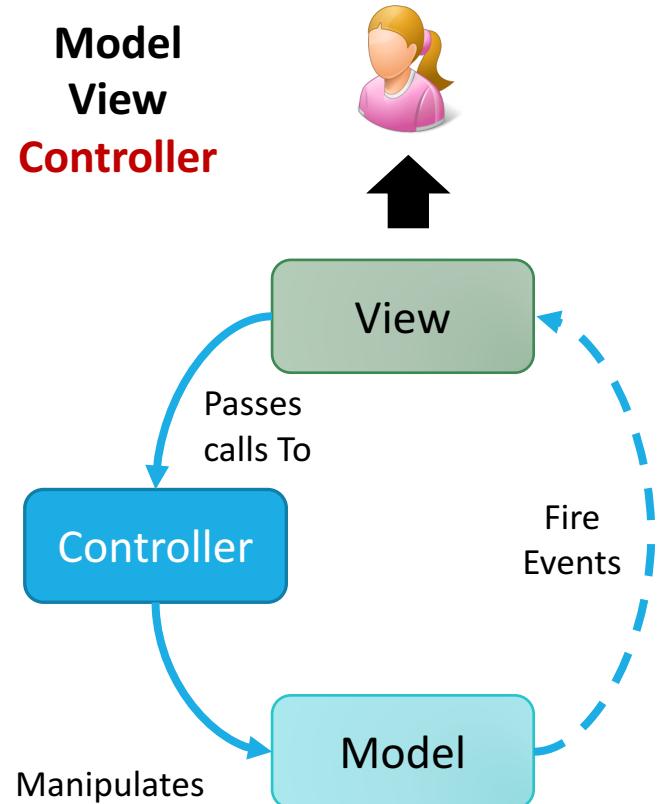
<http://www.slideshare.net/arafkarsh/functional-reactive-programming-64780160>



<http://www.slideshare.net/arafkarsh/function-point-analysis-65711721>

---

# UI Design Patterns



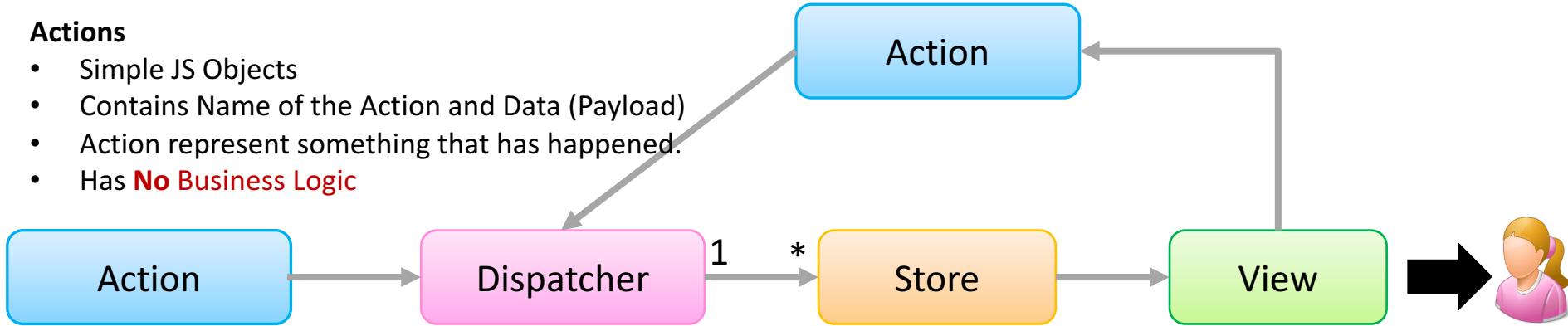
- The **Controller** is responsible to process incoming requests. It receives input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Typically, it acts as the coordinator between the View and the Model.

- The **Presenter** is responsible for handling all UI events on behalf of the view. This receive input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Unlike view and controller, **view and presenter are completely decoupled from each other's and communicate to each other's by an interface**. Also, presenter does not manage the incoming request traffic as controller.
- **Supports two-way data binding between View and ViewModel.**

- The View Model is responsible for exposing methods, commands, and other properties that helps to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.
- There is **many-to-one** relationship between View and ViewModel means many View can be mapped to one ViewModel.
- **Supports two-way data binding between View and ViewModel.**

### Actions

- Simple JS Objects
- Contains Name of the Action and Data (Payload)
- Action represent something that has happened.
- Has **No Business Logic**



**Every action is sent to all Stores** via callbacks the stores register with the Dispatcher

### Dispatcher

- Single Dispatcher per Application
- Manages the Data Flow View to Model
- Receives Actions and dispatch them to Stores

### Stores

- Contains state for a Domain (Vs. Specific Component)
- In Charge of modifying the Data
- Inform the views when the Data is changed by emitting the Changed Event.

### Controller-Views

- Listens to Store changes
- Emit Actions to Dispatcher

## Flux Core Concepts

1. One way Data Flow
2. No Event Chaining
3. Entire App State is resolved in store before Views Update
4. Data Manipulation ONLY happen in one place (Store).

UI Design Patterns  
Flux / Redux

# Redux Core Concepts

1. One way Data Flow
2. No Dispatcher compared to Flux
3. Immutable Store

Available for React & Angular

## Store

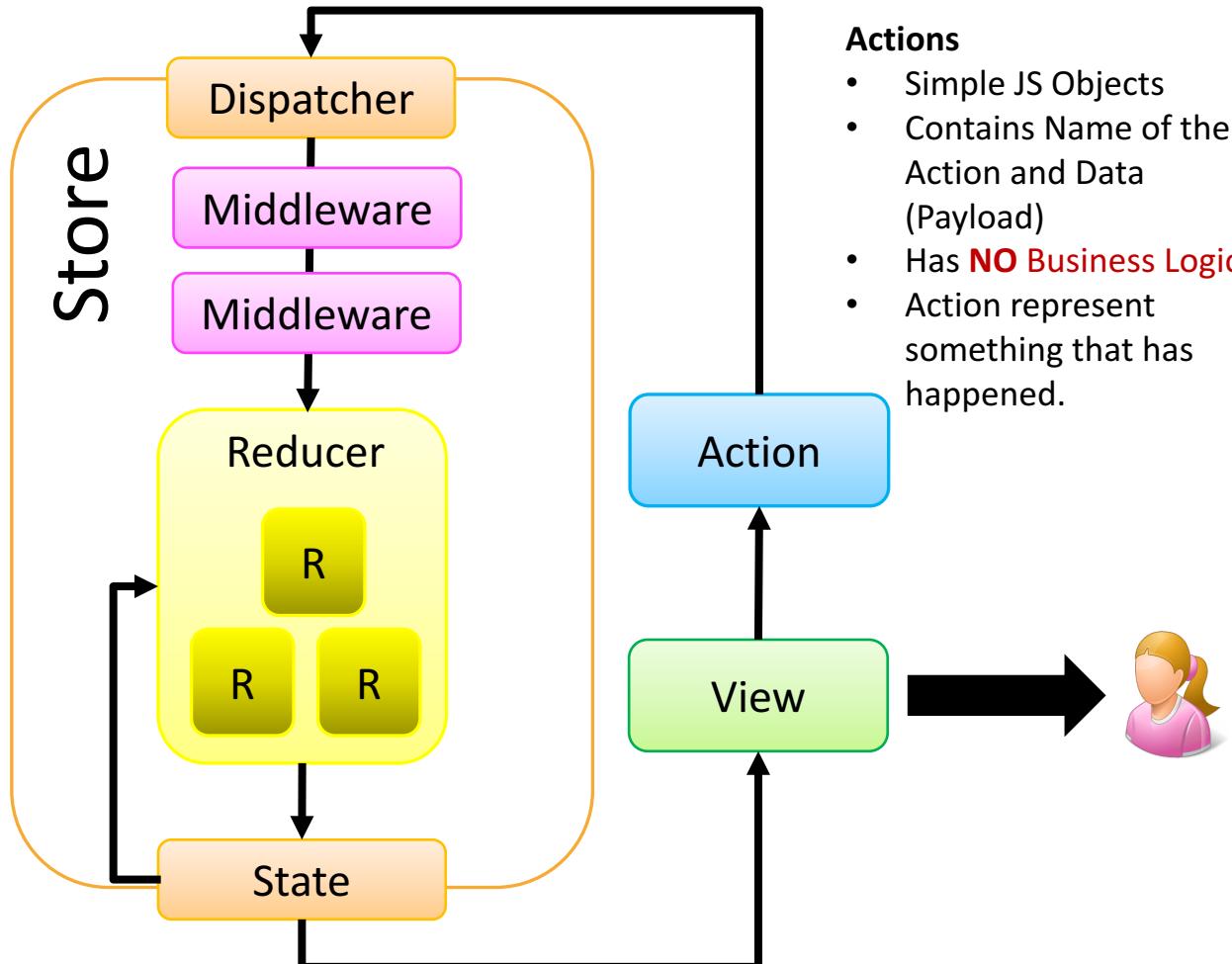
- Multiple View layers can **Subscribe**
- View layer to **Dispatch** actions
- Single Store for the Entire Application
- Data manipulation logic moves out of store to Reducers

## Middleware

- Handles External calls
- Multiple Middleware's can be chained.

## Reducer

- **Pure JS Functions**
- No External calls
- Can combine multiple reducers
- A function that specifies how the state changes in response to an Action.
- Reducer does **NOT modify** the **state**. It returns the **NEW State**.



UI Design Patterns Redux